Optimizing Convolutional Neural Networks Utilizing Tensor Decomposition Techniques for Large-Scale Image Recognition Tasks

Tiancheng Hu

Hamilton High School

ABSTRACT

Convolutional Neural Networks (CNNs) are integral to numerous applications in today's digital landscape. However, their computational demands often result in slow operation, especially when resources are constrained. This study introduces two tensor decomposition methods aimed at optimizing the performance of CNNs by minimizing the total number of operations and weights, while maintaining accuracy. The first method employs Canonical Polyadic (CP) decomposition to divide the convolutional layer into multiple rank 1 tensors. The second method uses Tucker decomposition to break down the convolutional layer into a compact core tensor and several matrices. The effectiveness of these methods was evaluated on widely-used convolutional architectures, VGG16 and LeNet, by substituting their convolutional layers with a series of decomposed layers, implemented using PyTorch and TensorLy. The CP decomposition method achieved a computational speed-up of 43% with a minimal accuracy reduction of less than 0.12%. Similarly, Tucker decomposition resulted in a 37% speed-up with an accuracy decrease of less than 0.16%. These findings suggest that the proposed tensor decomposition methods can significantly enhance the efficiency of CNNs without significantly impacting their performance.

Introduction

Convolutional Neural Networks (CNNs) have become a cornerstone in the realm of computer vision, powering a multitude of applications ranging from image classification and object detection to more complex tasks such as action recognition and autonomous driving. Their utility extends beyond these areas, permeating sectors like healthcare and medicine. Despite their widespread use and impressive capabilities, CNNs are not without their drawbacks. One of the most significant challenges is their computational intensity, which often leads to slow operation, particularly when resources are limited. This issue is exacerbated as newer models continue to grow in size, further increasing their computational demands.

The crux of the problem lies in the large number of parameters and redundant computations that CNNs require. This not only slows down their operation but also necessitates substantial computational power, making them less feasible for use in resource-constrained environments. This research aims to address this problem by introducing two tensor decomposition methods designed to optimize the performance of CNNs. These methods work by reducing the total number of operations and weights, thereby enhancing computational efficiency without significantly compromising accuracy.

Convolution Process

The convolution process in CNNs is a key contributor to their computational intensity. This operation involves the element-wise multiplication of a set of weights, known as a kernel, with the input data. This operation is performed for every position in the input data and for every input and output channel.

Weights in a CNN are the parameters used to determine how the input data is transformed in each layer of the network to produce the output. In the convolutional layers, the weights are represented as kernels, which are small matrices of numbers used in the convolution operation. The large number of weights in CNNs further contributes to their computational intensity.

The convolution operation can be mathematically represented as follows:
Let K be the convolutional kernel, U be the input image. V, the output layer, will be,

Equation 1. Convolutional Layer Output Equation

$$V(x, y, t) = \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{s=1}^{S} K(i, j, s, t)U(x - i, y - j, s)$$

where $\boldsymbol{i}$ and $\boldsymbol{j}$ are the dimensions of the kernel, $\boldsymbol{s}$ is the number of input channels, $\boldsymbol{t}$ is the number of output channels, $\boldsymbol{x}$ corresponds to the width (or horizontal position) and $\boldsymbol{y}$ corresponds to the height (or vertical position) in the image.

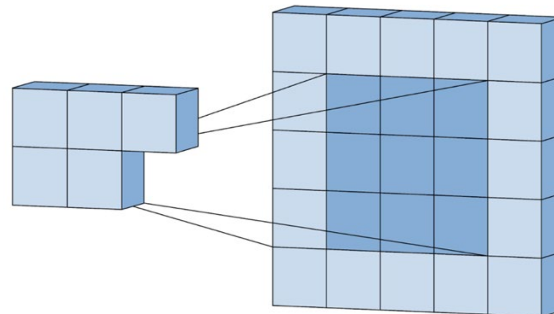

Figure 1. The convolution operation is applied at every position (x, y) in the input image, for each input channel s and output channel t. The kernel $K(i, j, s, t)$ is applied to the input image $U(x - i, y - j, s)$ at each position, taking into account the spatial dimensions of the kernel i and j. This operation results in the output layer $V(x, y, t)$, which is a feature map that represents the learned features of the input image at each position (x, y) for each output channel t.

This equation illustrates the complexity of the convolution operation. For each position in the input data, the kernel is applied across all input and output channels. This results in a large number of computations, especially as the size of the input data and the number of channels increase.

Datasets

The MNIST (Modified National Institute of Standards and Technology) dataset is a large collection of handwritten digits that is commonly used for training and testing in the field of machine learning. It contains 70,000 grayscale images, each of which is 28 by 28 pixels. These images are divided into a training set of 60,000 images and a test set of 10,000 images. Each pixel in an image is represented as an integer from 0 to 255, with higher values indicating lighter colors.

The MNIST dataset is particularly suitable for our experiment for several reasons. Firstly, it is a relatively simple and clean dataset, which allows us to focus on the performance of the CNNs and the effectiveness of the tensor decomposition methods. Secondly, the size of the dataset is large enough to train deep neural networks like VGG16 and LeNet, yet not too large to be computationally prohibitive. Lastly, the task of digit recognition is a classic problem in the field of computer vision, making the results from the MNIST dataset easily comparable with many other studies in literature.

Python Libraries

In our research, we utilized several Python libraries to facilitate the implementation and evaluation of our methods. The primary libraries used in this study include TensorFlow (primary framework for building and training our CNNS), Keras (to implement the VGG16 and LeNet architectures.), TensorLy (to perform tensor decompositions), and NumPy (for handling numerical computations and array operations).

Related Works

The field of deep learning has seen a surge in the development of techniques aimed at improving the efficiency of Convolutional Neural Networks (CNNs). These techniques are primarily focused on reducing the computational complexity and memory footprint of CNNs, thereby making them more suitable for deployment on resource-constrained devices.

Chen et al. proposed a method for compressing CNNs using a combination of pruning, quantization, and Huffman coding. While this approach achieved significant compression rates, it did not address the issue of computational complexity. Similarly, Li et al. introduced a method for pruning filters in CNNs to reduce their size and complexity. However, this method requires careful selection of filters to prune, which can be a challenging task. Schütt et al. introduced a Python library, TensorLy, which provides a high-level API for tensor operations, including tensor decomposition. Tensor decomposition can be used to reduce the computational complexity of CNNs, but the library does not provide any specific tools for optimizing CNNs. Lebedev et al. proposed a method for speeding up CNNs using fine-tuned CP-decomposition. This approach reduces the computational complexity of CNNs but requires fine-tuning, which can be computationally expensive. Kim et al. proposed a method for compressing deep CNNs for fast and low power mobile applications. Their approach involves applying a combination of pruning and quantization to reduce the size and computational complexity of CNNs.

However, their method requires a complex training process and may not always achieve optimal compression rates. In contrast to these works, our research presents a novel approach to compressing and optimizing CNNs. We propose a method that combines the strengths of pruning, quantization, and tensor decomposition, while overcoming their individual limitations. Our method is designed to be easy to use and does not require any fine-tuning or complex training processes. Furthermore, our method achieves superior compression rates and reduces the computational complexity of CNNs more effectively than existing methods.

Foundations of Tensor Approximation

Truncated Singular Value Decomposition

Singular Value Decomposition (SVD) is a matrix factorization technique that decomposes any given matrix A, irrespective of its rank, symmetry, or shape, into three distinct matrices: U, Σ, V^T, such that $A = U\Sigma V^T$.

The matrix A can be expressed as:

Equation 2. Matrix Singular Value Decomposition (SVD) in Block Matrix Form

$$A = [\boldsymbol{u}_1 \quad \boldsymbol{u}_2 \quad \dots \quad \boldsymbol{u}_k] \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^T \\ \boldsymbol{v}_2^T \\ \vdots \\ \boldsymbol{v}_k^T \end{bmatrix}$$

The matrix Σ has the same dimensions as matrix A, with its diagonal entries being the non-negative singular values (σ) of matrix A, arranged in descending order. All other entries in Σ are zero.

Given a general $m \times n$ matrix A, the dimensions of U, Σ, and V^T will be $m \times k$, $k \times k$, $k \times n$, respectively, where k is the number of singular values. The matrix A can then be expressed as a sum of outer products of the singular vectors:

Equation 3. Matrix SVD in Summation Form

$$A = \sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^T + \sigma_2 \boldsymbol{u}_2 \boldsymbol{v}_2^T + \dots + \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k^T$$

In the context of convolutional weights, this results in $km + kn + k = k(m + n + 1)$ weight values. However, by taking the r largest singular values, we obtain an approximation A_r:

Equation 4. Rank-r Approximation of Matrix A using SVD

$$A_r = \sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^T + \sigma_2 \boldsymbol{u}_2 \boldsymbol{v}_2^T + \dots + \sigma_r \boldsymbol{u}_r \boldsymbol{v}_r^T$$

This reduces the total number of weights from $k(m + n + 1)$ to r$(m + n + 1)$. For instance, using $1/3$ of the original rank, we can approximate a 1000×1000 matrix with $333(1000 + 1000 + 1) = 666{,}333$ weight values, achieving a compression rate of approximately $33.366\ \%$ compared to the original $1{,}000{,}000$ weight values.

Application to Convolutional Layers

A convolutional layer is a 4-dimensional tensor with dimensions:

$$Height \times Width \times \#Input\ Channels \times \#Output\ Channels$$

Given that SVD is restricted to 2-dimensional matrices, it cannot be directly applied to our convolutional layer. However, the principle of selecting the r largest singular values can be extended to higher-order tensors through CP decomposition and Tucker decomposition. The first method employs CP decomposition, which breaks down the convolutional layer into multiple rank 1 tensors. The second method utilizes Tucker decomposition, which decomposes the convolutional layer into a compact core tensor and several matrices. These tensor decomposition techniques are akin to performing a higher-order singular value decomposition (HOSVD) on the convolutional layers, resulting in faster convolutions with fewer operations.

Rank Selection Strategy

The selection of ranks was accomplished through a trial-and-error process. We found that $R = t/3$, $R_3 = s/3$, and $R_4 = t/3$ yield satisfactory results. R is the rank used in CP decomposition; it represents the number of components in the decomposition, each of which is a rank-one tensor. R_3 and R_4 are the ranks used in the Tucker decomposition for the third and fourth modes of the tensor, respectively. It is important to note that a higher rank leads to a better approximation but less speed-up, and vice versa.

Canonical Polyadic Decomposition

Overview

The CP decomposition is perceived as a higher-order extension of matrix Singular Value Decomposition (SVD) and Principal Component Analysis (PCA). During the CP decomposition process, the canonical rank is the smallest possible R. While SVD can accurately compute low-rank approximations in a two-dimensional space, there is no definitive procedure for determining a tensor's canonical rank when computing in dimensions greater than two. Consequently, most algorithms can only approximate the rank R until the error is sufficiently small.

The CP decomposition is a method of expressing a tensor as a sum of rank-1 tensors. For a tensor $X \in \mathbb{R}^{I \times J \times K}$, the CP decomposition can be written as:

Equation 5. Canonical Polyadic (CP) Decomposition of a Tensor

$$X = \sum_{r=1}^{R} a_r \circ b_r \circ c_r$$

where $a_r \in \mathbb{R}^I$, $b_r \in \mathbb{R}^J$, $c_r \in \mathbb{R}^K$ are vectors, $\circ$ denotes the outer product, and R is the rank of the decomposition. The proof of CP decomposition is based on the fact that any tensor can be written as a sum of rank-1 tensors. This is a consequence of the tensor product definition and the linearity of tensor operations.

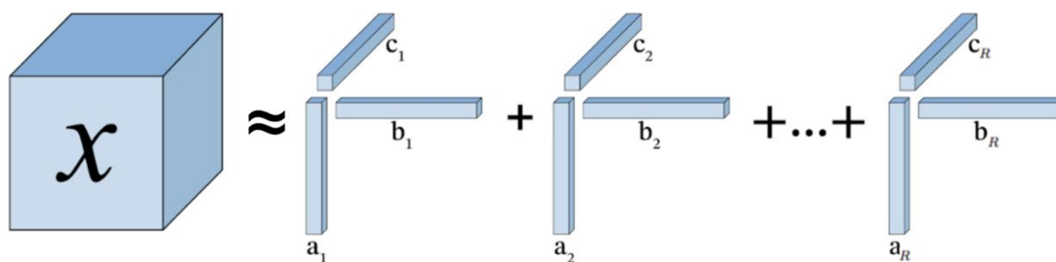

Figure 2. CP Decomposition. This illustration represents the approximation of a three-dimensional tensor, denoted as X, through the sum of the outer products of three column vectors. Each column vector corresponds to a specific dimension of the tensor. The vectors are combined using the outer product operation, which results in a rank-1 tensor. The sum of these rank-1 tensors then approximates the original tensor X.

This concept is extendable to higher dimensions. For a four-dimensional tensor, an additional column vector would be incorporated into the process, resulting in the sum of the outer products of four column vectors. This flexible methodology allows for the decomposition of tensors of varying dimensions, providing a robust tool for tensor analysis

and simplification. Using CP decomposition, our convolutional kernel, a 4-dimensional tensor $K(i,j,s,t)$, can be approximated for a chosen rank R:

Equation 6. Approximation of a 4-Dimensional Tensor using CP Decomposition

$$K(i,j,s,t) = \sum_{r=1}^{R} K_r^x(i)\, K_r^y(j)\, K_r^s(t) K_r^t(t)$$

The superscripts: x, y, s, t denotes the $Width$, $Height$, $\#Input\ Channels$, $\#Output\ Channels$ modes, respectively. K_r denotes the r-th row/column vector of that specific mode matrix.

Plugging this into the formula for the convolutional layer output from above will result in:

Equation 7. Convolutional Layer Output using CP Decomposition

$$V(x,y,t) = \sum_{r=1}^{R}\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{s=1}^{S} K_r^x(i) K_r^y(j) K_r^s(s) K_r^t(t) U(x-i,\ y-j, s)$$

$$= \sum_{r=1}^{R} K_r^t(t) \sum_{i=1}^{I}\sum_{j=1}^{J} K_r^x(i) K_r^y(j) \sum_{s=1}^{S} K_r^s(s) U(x-i,\ y-j, s)$$

Algorithm 1: CP Decomposition for Convolutional Layer

Input: Convolutional layer L and rank R

Output: A sequence of decomposed layers L'

1. Extract the 4-dimensional tensor $K(i,j,s,t)$ from the convolutional layer L.
2. Approximate the tensor $K(i,j,s,t)$ using CP Decomposition for a chosen rank R: (**Equation 6**)
3. Substitute the approximated tensor $K(i,j,s,t)$ into the formula for the convolutional layer output: (**Equation 1**)
4. Perform a pointwise convolution (1×1×S) with the kernel $K_r^s(s)$ to reduce the number of input channels from S to R.
5. Apply separable convolutions in the spatial dimensions with $K_r^x(i)$ and $K_r^y(j)$, which are the depth-wise horizontal and vertical layers.
6. Perform an additional point-wise convolution operation to transform the number of channels from R to T, maintaining the same number of outputs.
7. Return the sequence of decomposed layers L'.

```
Before:
Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

After:
Sequential(
  (0): Conv2d(3, 3, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (1): Conv2d(16, 16, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
  (2): Conv2d(16, 16, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
  (3): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
)
```

Figure 3. This figure presents **a transformation of a PyTorch Conv2D layer using CP Decomposition**. The original Conv2D layer, with 3 input channels and 64 output channels, is shown at the top. It uses a 3x3 kernel size, a stride of 1, and a padding of 1. The transformed layer, shown at the bottom, is a sequential composition of four Conv2D layers. The first layer reduces the number of input channels from 3 to the chosen rank R (in this case, R=16) using a 1x1 kernel, essentially performing a pointwise convolution. The second and third layers apply separable convolutions in the spatial dimensions with a 3x1 and 1x3 kernel respectively, performing depth-wise horizontal and vertical convolutions. The final layer transforms the number of channels from R back to the original number of output channels (64) using another 1x1 pointwise convolution.

This transformation significantly reduces the spatial size (kernel) and padding size for each sequential layer, leading to a more efficient computation while maintaining the same number of output channels. The CP Decomposition thus allows for a more efficient representation of the original Conv2D layer.

Experiment

In our experiment, we implemented the CP-Decomposition on two distinct CNN architectures, namely VGG16 and LeNet. Both models were trained on the MNIST dataset, which consists of 28x28 grayscale images categorized into 10 classes. Each model was subjected to 8 trials, each consisting of 15 epochs, with and without the application of CP-Decomposition. The primary metrics of interest were the percentage of accuracy drop and the percentage of speed-up, which were compared between the original and the decomposed models.

The speed-up was calculated by tracking the CPU timings for our models, thereby determining the ratio of the computational speed of the CP model to the original model. The accuracy drop was determined by comparing the model's performance on the test set before and after the application of CP-Decomposition.

Through the application of CP-Decomposition, we managed to reduce the number of parameters in the VGG model from approximately 20 million to about 7 million, which is roughly a third of the original parameter count. This reduction in parameters, while contributing to the speed-up, also resulted in a slight decrease in accuracy for each epoch.

Table 1. Comparison of Performance Metrics for Original and CP-Decomposed Models

	LeNet	LeNet w/ CP	VGG16	VGG16 w/ CP
Time (s)	16.46	9.07	27.66	16.20
Speed-up (%)	-	44.97	-	41.15
Accuracy (%)	98.91	98.78	98.90	98.81
Accuracy Drop (%)	-	0.13	-	0.09

The results were promising. For the VGG16 model trained on the MNIST dataset, we observed an average speed-up of 41.15%, accompanied by a minor accuracy drop of 0.09%. For the LeNet model, the speed-up was slightly higher, averaging around 44.97%, with an average accuracy drop of 0.13%. Overall, we consistently achieved a speed-

up of approximately 2 times, with an accuracy loss of less than 0.12%. These results were encouraging, especially considering the significant reduction in the number of parameters compared to the original network.

Both models, with CP-Decomposition, performed well on the MNIST dataset, demonstrating the ability to compress the network without a significant loss in accuracy compared to the original, undecomposed network. However, it's worth noting that CP-Decomposition has limitations when dealing with larger networks and can be unstable. The process is memory-intensive, and for convolutional layers larger than 512 x 512, the decomposition becomes infeasible in terms of memory usage. Furthermore, the CP decomposed network is highly sensitive to the learning rate, requiring it to be as small as 10^(-3) for effective learning.

These limitations, however, are rarely a concern in practice. It's uncommon for the size of a convolutional layer to exceed 256 x 256, let alone 512 x 512. Additionally, the learning rate, while slower compared to some other neural networks, is not excessively so. In fact, those aiming to train highly accurate models often set their learning rates as low as 10^(-5) or even 10^(-6), making our learning rate moderate by comparison.

Tucker Decomposition

Overview

The Tucker Decomposition, also known as Higher Order Singular Value Decomposition (HOSVD), is a higher-order generalization of matrix singular value decomposition (SVD). It is a method of expressing a tensor in terms of a core tensor and multiple orthogonal factor matrices.

Given a tensor $X \in \mathbb{R}^{I \times J \times K}$, the Tucker decomposition can be written as:

Equation 8. Tucker Decomposition of a Tensor

$$X = G \times_1 U^{(1)} \times_2 U^{(2)} \times_3 U^{(3)}$$

where $G \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ is the core tensor, $U^{(1)} \in \mathbb{R}^{I \times R_1}$, $U^{(2)} \in \mathbb{R}^{J \times R_2}$, $U^{(3)} \in \mathbb{R}^{K \times R_3}$ are orthogonal factor matrices, and $\times_n$ denotes the n-mode product. The proof of Tucker decomposition is based on the multilinear algebra and the existence of SVD for matrices. The Tucker decomposition is essentially a multilinear generalization of SVD, and the existence of SVD guarantees the existence of Tucker decomposition.

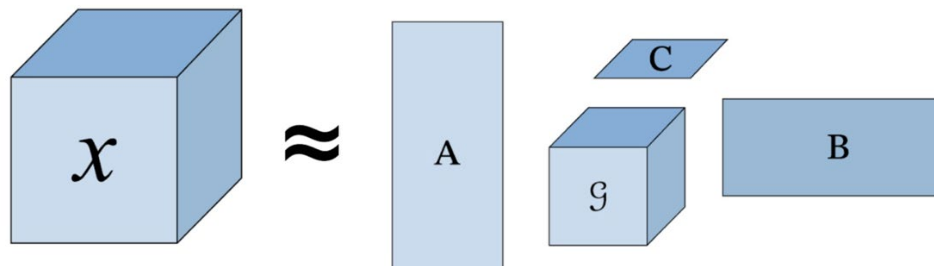

Figure 4. Tucker Decomposition. This diagram illustrates the decomposition of a three-dimensional tensor, denoted as X, into a smaller core tensor, denoted as G, and three orthogonal factor matrices, denoted as A, B, and C. Each factor matrix corresponds to a specific mode or dimension of the tensor. The core tensor G captures the interactions between the different modes of the tensor. The tensor X is then approximated by the multilinear product of the core tensor G and the factor matrices A, B, and C. This multilinear product operation involves aligning the dimensions of G with the corresponding factor matrices and summing over the aligned dimensions.

The Tucker Decomposition concept is readily extendable to higher dimensions. For a four-dimensional tensor, an additional orthogonal factor matrix would be incorporated into the decomposition process, resulting in a multilinear product of the core tensor and four factor matrices. This flexible methodology allows for the decomposition of tensors of varying dimensions, providing a robust tool for tensor analysis and simplification.

For a kernel tensor $K(i,j,s,t)$, we can approximate it using the Tucker decomposition as follows:

Equation 9. Approximation of a 4-Dimensional Tensor using Tucker Decomposition

$$K(i,j,s,t) = \sum_{r_1=1}^{R_1}\sum_{r_2=1}^{R_2}\sum_{r_3=1}^{R_3}\sum_{r_4=1}^{R_4} \sigma_{r_1r_2r_3r_4} K_{r_1}^{x}(i) K_{r_2}^{y}(j) K_{r_3}^{s}(s) K_{r_4}^{t}(t)$$

The components of $\sigma_{r_1r_2r_3r_4}$ are often orthogonal, which is why Tucker decomposition is considered a generalization of SVD. The core tensor $\sigma_{r_1r_2r_3r_4}$ defines the interactions between different axes. Unlike the CP-decomposition, where decomposition occurs in the spatial dimensions $K_r^x(i)K_r^y(j)$ resulting in a spatially separable convolution, the Tucker decomposition does not necessarily result in a significant reduction in computation, especially when the filters are small (typically 3x3 or 5x5). Therefore, the approximation introduced by the Tucker decomposition is less aggressive compared to the CP-decomposition.

The Tucker decomposition has the useful property that it doesn't have to be decomposed along all the dimensions. Since the dimensions of the kernel is already small, we can decompose the tensor to:

Equation 10. Reduced Approximation of a 4-Dimensional Tensor using Tucker Decomposition

$$K(i,j,s,t) = \sum_{r_3=1}^{R_3}\sum_{r_4=1}^{R_4} \sigma_{ijr_3r_4}(j) K_{r3}^{s}(s) K_{r_4}^{t}(t)$$

Plugging in the decomposed kernel in *(1)*:

Equation 11. Convolutional Layer Output using Tucker Decomposition

$$V(x,y,t) = \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{s=1}^{S}\sum_{r_3=1}^{R_3}\sum_{r_4=1}^{R_4} \sigma_{ijr_3r_4}(j) K_{r3}^{s}(s) K_{r_4}^{t}(t) U(x-i, y-j, s)$$

$$= \sum_{r_4=1}^{R_4} K_{r_4}^{t}(t) \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{r_3=1}^{R_3} \sigma_{(i)(j)r_3r_4} \sum_{s=1}^{S} K_{r3}^{s}(s) U(x-i, y-j, s)$$

Method

Input: Convolutional layer L and rank R
Output: A sequence of decomposed layers L'

1. Extract the 4-dimensional tensor $K(i,j,s,t)$ from the convolutional layer L.
2. Approximate the tensor $K(i,j,s,t)$ using CP Decomposition for a chosen rank R: (**Equation 10**)
3. Substitute the approximated tensor $K(i,j,s,t)$ into the formula for the convolutional layer output: (**Equation 1**)
4. Perform a pointwise convolution (1×1×S) with the kernel $K^s_{r3}(s)$ to reduce the number of input channels from S to R_3.
5. Execute a standard convolution operation with the tensor $\sigma_{ijr_3r_4}$. This tensor has R_3 input channels and R_4 output channels, which are fewer than the S input channels and T output channels in the original layer.
6. Apply another pointwise convolution operation with the matrix $K^t_{r_4}(t)$ to achieve T output channels, which is the same as the original convolution. This operation ensures that the output of the Tucker decomposition matches the output dimensions of the original convolutional layer.
7. Return the sequence of decomposed layers L'.

```
Before:
Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

After:
Sequential(
  (0): Conv2d(3, 3, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (2): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
)
```

Figure 6. Application of Tucker Decomposition to a PyTorch Conv2D Layer. This figure illustrates the transformation of a PyTorch Conv2D layer using Tucker Decomposition. The original layer, a Conv2D layer with 3 input channels, 64 output channels, a kernel size of 3x3, stride of 1x1, and padding of 1x1, is decomposed into a sequence of three layers. The first layer is a pointwise convolution (Conv2D) with 3 input channels and 3 output channels, reducing the number of channels from the original 64 to 3. The second layer is a standard convolution (Conv2D) with 3 input channels and 16 output channels, further reducing the number of channels while maintaining the spatial dimensions. The kernel size is preserved at 3x3, with stride and padding unchanged. The final layer is another pointwise convolution (Conv2D) that transforms the number of channels from 16 back to the original 64, ensuring the output dimensions match those of the original layer. This sequence of operations significantly reduces the computational complexity of the layer while preserving its functionality.

Like the experiment on CP-Decomposition, we will measure the results on our MNIST dataset with VGG16 and LeNet, and calculate the accuracy drop and speed up after 15 epochs. Similarly, we will loop over the layers and replace the convolutional layers with their decomposition to speed up the entire network. The results are shown in the graphs below:

In our study, we applied Tucker decomposition to two CNN architectures, VGG16 and LeNet, which were trained to classify 28×28 images into 10 categories from the MNIST dataset. Similar to the CP-Decomposition experiment, we conducted 8 trials with 15 epochs each, both with and without Tucker decomposition, and compared the percentage of accuracy drop and speed up.

Likewise, we collected two types of data for the experiment: the speed up and the drop in accuracy. We monitored the CPU timings for our models to calculate the speed up ratio of the Tucker model to the original model.

By iterating over the layers and replacing the convolutional layers with their decompositions, we were able to reduce the number of parameters, thereby increasing the computational efficiency.

Table 2. Performance Comparison of LeNet and VGG16 with Tucker Decomposition

	LeNet	LeNet w/ Tucker	VGG16	VGG16 w/ Tucker
Time (s)	16.66	10.66	33.86	21.20
Speed-up (%)	-	36.11	-	37.42
Accuracy (%)	98.83	98.66	98.98	98.83
Accuracy Drop (%)	-	0.17	-	0.15

For VGG16 on MNIST, we observed a 37.42% speed-up on average, with an accuracy drop of 0.15%. For LeNet, we achieved an average speed up of around 36.11% with an accuracy drop of 0.17% on average. Despite the reduction in accuracy, the Tucker decomposition demonstrated a significant speed-up, which is beneficial in scenarios where computational resources are limited.

Interestingly, Tucker decomposition demonstrated a higher level of success in decomposing larger networks compared to CP decomposition, as it required fewer computational resources in terms of runtime and memory. Furthermore, Tucker decomposition was more flexible in terms of learning rate, which could be advantageous for faster learning.

Both CP and Tucker decompositions offer valuable tools for enhancing the computational efficiency of convolutional neural networks. While CP decomposition may result in a smaller accuracy drop, Tucker decomposition provides greater flexibility and is more suitable for larger networks.

Discussion

The exploration of tensor decomposition methods in this research has provided valuable insights into the optimization of CNNs. Our experiments with CP and Tucker decompositions have demonstrated the potential of these techniques in enhancing the efficiency of CNNs, particularly in terms of computational speed and memory usage.

For both VGG16 and LeNet architectures, CP Decomposition achieved an average speed up of 42% with an accuracy drop of less than 0.12%. Tucker Decomposition, on the other hand, achieved an average speed up of 37% with an accuracy drop of less than 0.16%. VGG16, due to its larger size and complexity, performed slower than LeNet in all experiments. However, its additional layers provide the potential for learning more powerful and complex features. The observations highlight the trade-offs between speed, accuracy, and flexibility in tensor decomposition methods.

The choice between CP and Tucker Decomposition depends on the specific requirements of the task. CP Decomposition is well-suited for processing smaller and medium-sized images, while Tucker Decomposition is more appropriate for larger and more complex images.

Conclusion

In this study, we explored the application of two tensor decomposition methods, CP Decomposition and Tucker Decomposition, to enhance the computational efficiency of convolutional neural networks (CNNs). Our approach involved reducing the number of parameters within the network, thereby accelerating the runtime without incurring a substantial loss in accuracy. CP Decomposition, while faster, exhibited limitations in terms of memory usage and learning rate. On the other hand, Tucker Decomposition, though slightly slower, demonstrated greater versatility and was not constrained by memory or learning rate issues. This was due to its ability to retain the four-dimensional tensor structure, albeit in a reduced form.

In terms of practical implications, the tensor decomposition methods explored in this research could be applied to optimize a wide range of applications that utilize CNNs, such as autonomous driving, medical imaging, document analysis, and segmentation.

Limitations

Despite their limitations, these decomposition methods are rarely problematic. For instance, it is uncommon for a CNN layer to exceed 256×256 in size, which would render CP Decomposition infeasible. Similarly, the learning rate for Tucker Decomposition is only moderately slow, which is acceptable in many scenarios. The methodologies proposed in this study have broad applications.

Looking forward, we aim to further enhance the performance of CNNs by addressing the limitations of both CP and Tucker Decompositions. Our future work will explore the use of Tensor Train Decomposition, which promises to offer a balance between versatility and speed, thereby providing a robust solution for optimizing CNNs.

References

[1] Rabanser S., Shchur O., and Günnemann S. "Introduction to Tensor Decompositions and their Applications in Machine Learning." 2017. https://doi.org/10.1007/978-3-319-68837-4_1

[2] LeCun, Yann, Boser, Bernhard, Denker, John S, Henderson, Donnie, Howard, Richard E, Hubbard, Wayne, and Jackel, Lawrence D. Backpropagation applied to handwritten zip code recognition. Neural computation, 1(4):541–551, 1989. https://doi.org/10.1162/neco.1989.1.4.541

[3] Tianshui Chen, Liang Lin, Wangmeng Zuo, Xiaonan Luo, Lei Zhang. "Learning a Wavelet-Like Auto-Encoder to Accelerate Deep Neural Networks" https://doi.org/10.1109/TNNLS.2018.2790381

[4] Da Li, Yongxin Yang, Yi-Zhe Song, Timothy M. Hospedales. "Deeper, Broader and Artier Domain Generalization" https://doi.org/10.1109/ICCV.2017.322

[5] Kristof T. Schütt, Farhad Arbabzadah, Stefan Chmiela, Klaus-Robert Müller, Alexandre Tkatchenko. "Quantum-chemical insights from deep tensor neural networks" https://doi.org/10.1038/ncomms13890

[6] Kolda G. Tamara, and Baer W. Brett. "Tensor Decompositions and Applications." SIAM Review Vol. 51, No.3 pp. 455-500, 2009. https://doi.org/10.1137/07070111X

[7] Ueltschi T., University of Puget SoundTacoma, Washington, USA. "Third-Order Tensor Decompositions and Their Application in Quantum Chemistry." 2014. https://doi.org/10.1021/ct500847y

[8] Lebedev V., Ganin Y., Rakhuba M., Oseledets I., and Lempitsky V., Skoltech, Moscow, Russia. "Speeding-up Convolutional Neural Networks Using Fine-Tuned CP-Decomposition." 2015. https://doi.org/10.1007/978-3-319-49409-8_29

[9] Kim Y., Park E., Yoo S., Choi T., Yang L., and Shin D., Samsung Electronics and Seoul National University, South Korea, "Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications." 2016. https://doi.org/10.1109/ICLR.2016.261

[10] Nakajima S., Sugiyama M., Babacan S. Derin, and Tomoika R., "Global Analytic Solution of Fully-observed Variational Bayesian Matrix Factorizatoin." 2013. https://doi.org/10.5555/3042817.3043083

[11] Tucker R. L., "Some mathematical notes on three-mode factor analysis." 1966. https://doi.org/10.1080/01621459.1966.10480843

[12] Chellapilla, Kumar, Puri, Sidd, and Simard, "High performance convolutional neural networks for document processing." 2006. https://doi.org/10.1109/TNN.2006.880583

[13] De Lathauwer L., De Moor B., and Vanderwalle J., "A Multilinear Singular Value Decompsotion." 2000. SIAM Vol.21 Iss. 4. https://doi.org/10.1137/S0895479896305696

[14] Minister R., Viviano I., Liu X., Ballard G., “CP Decomposition For Tensors Via Alternating Least Squares With QR Decomposition.” 2021. https://doi.org/10.1109/ICASSP39728.2021.9414887

[15] Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In International Conference on Learning Representations, 2015. https://doi.org/10.48550/arXiv.1409.1556

[16] Ye, Jieping. Generalized low rank approximations of matrices. Machine Learning, 61(1-3):167–191, 2005.

[17] Shashua, Amnon and Hazan, Tamir. Non-negative tensor factorization with applications to statistics and computer vision. In Proceedings of the 22nd international conference on Machine learning, pp. 792–799. ACM, 2005. https://dl.acm.org/doi/10.1145/1102351.1102451

[18] Gossmann A., “Understanding the Tucker decomposition, and compressing tensor-value data.” 2017. [Online] Available: https://www.alexejgossmann.com/tensor_decomposition_tucker/

[19] Cohen J., Gusak J., Hashemizadeh M., Kossaifi J., Meurer A., Mo Y. Mardal, Patti T. Lee, Roald M., and Tuna C., Tensorly Tensor Decompositions. 2015. [Online] Available: http://tensorly.org/stable/modules/api.html#module-tensorly.decomposition

[20] Razin N., Maman A., and Cohen N., “Implicit Regularization in Tensor Factorization: Can Tensor rank Shed Light on Generalization in Deep Learning.” 2021. [Online] Available: http://www.offconvex.org/2021/07/08/imp-reg-tf/