Low-Rank Computationally Efficient Convolutional Neural Networks

Code: https://github.com/srijaladi/CS131Project/tree/main

Sri Jaladi
Stanford Computer Science
sjaladi@stanford.edu

Ishan Khare
Stanford Computer Science
iskhare@stanford.edu

Bar Wiener
Stanford Computer Science
barw@stanford.edu

1. Introduction

The bottleneck of computational efficiency has made itself prevalent with the expanding adoption of AI in daily life. Whether this is with OpenAI spending millions of dollars each day to run ChatGPT or Meta buying close to a million GPUs, it is clear that computational efficiency is a high priority. To combat these issues, some work has been done towards creating more efficient models. We plan on similarly experimenting, but with convolutional neural networks (CNNs) and specifically the convolutional layers within these models. Given the high impact that CNNs have on the field of computer vision (CV), we believe our work is a strong step in making CV more efficient. Specifically, we plan on training CNNs while forcing their convolutional weight matrices to be low-rank, where rank refers to the rank of the weight matrix. We hypothesize this process should allow for the model, during test-time, to reduce computations (increasing speed), but may cause loss of some performance due to restricting the model's possible weight space. Our project will explore this tradeoff. Our problem statement is: **"How does utilizing low-rank convolutions affect a CNN's performance in computational time, memory, and accuracy?"**. We hope our work can help bring often hefty CV models and CNNs to daily-usable devices such as phones and computers.

2. Related Work

We note that a large portion of our project relies on heavy mathematical results and proofs. We note that these have each individually been done in literature and we are combining their results in order to generate our novel idea of low-rank convolutions. Due to limited space, we cannot fully prove out all these components, but we will appropriately cite papers which do these computations. First, we note that previous work has proven that a convolution $C \in \mathbb{R}^{nxn}$ applied onto a matrix X can actually be expressed entirely as a series of matrix multiplications $M_{1,a}M_{2,a}...M_{n,a}XM_{n,b}^{T}M_{(n-1),b}...M_{2,b}^{T}M_{1,b}^{T}$ where each $M_{i,a}, M_{i,b} : \forall i \in \{1, 2, 3...n\}$ are Toeplitz matrices [3]. Combining the terms $M_{i,a} : \forall i \in \{1, 2, 3, ...n\}$ and the terms $M_{i,b} : \forall i \in \{1, 2, 3, ...n\}$ yields just M_aXM_b, which is how our code does this process. A more explicit and detailed proof for the creation of each Toeplitz Matrix and multiplication process (along with corresponding code) can be found in another paper [6]. There has also been significant research in utilizing low-rank approximations (LoRA) to reduce parameters in neural networks. While most LoRA papers focus on LLMs and linear layers, one paper utilized a special rank-based algorithm to try and measure the importance of each parameter in a CNN model and then prune low-importance parameters [7]. Unfortunately, almost all the existing research in LoRA first trains the model regularly, then approximates the weight matrices, and then potentially does fine-tuning after [5] [4] [9]. While this has yielded promising results, this does not account for constraining the weight matrices to low-rank throughout the entirety of training. We believe that this will yield even better performance and accuracy. The concern with this approach, according to research, is that it may be more time-inefficient [5]. However, we hypothesize that the usage of matrix multiplications for CNNs will help reduce this issue. Further, we believe that maintaining (and potentially improving) testing time while reducing parameter counts and maintaining accuracy will be well worth this cost. However, we do test and emphasize the training time due to this concern. There is also a gap in low-rank training research in exploring applications within the realm of CNNs, which are a massive part of the computer vision space, and which is why we chose to explore this area.

3. General Methodology

Our idea comprises of two components. Firstly, we use the fact that convolutions on a matrix can instead be expressed as as a series of matrix multiplications. Next, we use the fact that the lower the rank of the convolution weight matrix, the lower the number of matrices there are in such a series. Therefore, reducing the rank of the convolution weight matrix reduces the number of computations that must be done. Thus, we trained CNN image classifiers on the CIFAR-10 dataset [1] while applying each of the model's convolutions via a series of matrix multiplications. This way, we can ensure that the rank of the convolution rank matrix is maintained throughout the entirety of training. We experiment with modifying the rank of the convolution weight matrices and viewing its impact on the number of calculations, time taken for model computations, and the performance or degradation of the model. Note that this guarantees higher space efficiency of order K where the kernel used during convolutions is of size K x K. Explicitly, convolutions would regularly utilize space $O(K^2)$, but under our scheme, they only utilize space of order $O(LK)$ where L is the rank to which we constrain our weight matrices to. When $L << K$, it becomes relatively negligible, and so we can effectively cut an order of K off the number of parameters we store.

4. Algorithmic Methodology

We now describe our algorithm to create a low-rank convolutional layer enforced onto a rank-N space. In order to do this, we create our own custom PyTorch module for a RankConv2d layer, which takes as input all the PyTorch Convolutional Layer inputs along with the desired rank of the convolution. During initialization of this layer, we need to ensure that the initialization of our low-rank convolution is approximately equal to the initialization process that PyTorch default convolution uses. To replicate the Xavier Initialization

used by PyTorch default convolution, we follow a 3 step process. First, we initialize a K x K matrix M, and initialize its values with Xavier Initialization. We then utilize the Eckert-Young Rank Approximation algorithm to approximate M to our layer's desired rank of L. To apply the Eckert-Young Algorithm, we run Singular Value Decomosition (SVD) on M to obtain $M = U\Sigma V^T$ [2]. We then obtain

$M_a = \left[\sqrt{\Sigma_{1,1}}col_U(1) \quad \sqrt{\Sigma_{2,2}}col_U(2) \quad ... \quad \sqrt{\Sigma_{L,L}}col_U(L)\right]$

$M_b^T = \left[\sqrt{\Sigma_{1,1}}col_V(1) \quad \sqrt{\Sigma_{2,2}}col_V(2) \quad ... \quad \sqrt{\Sigma_{L,L}}col_V(L)\right]$

where $col_A(i)$ represents the ith column in matrix A. This way, $M_a, M_b^T \in \mathbb{R}^{K \text{ x } L}$ and $M = M_a M_b$, ensuring that the convolution matrix M is rank L. Importantly, we take the square root of each value in Σ and apply it to each M_a and M_b equally so that the magnitudes of each of these matrices are equal. Note that this initialization scheme is vital to our project as we discovered that without this, the magnitudes always lead to either exploding or vanishing gradient problems. Once we have M_a, M_b, we create the Toeplitz matrices outlined in the related work of $\{M_{i,a} : \forall i \in \{1, 2, 3...L\}\}, \{M_{i,b} : \forall i \in \{1, 2, 3...L\}\}$. Note that $M_{i,a}$ represents the Toeplitz Matrix created by utilizing the ith column of matrix M_a. A visualization of a Toeplitz matrix from some vector x can be seen in Figure 1. We then apply the convolution onto our image X by

$$\begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n & 0 & 0 & 0 & \cdots & 0 \\ 0 & x_1 & x_2 & x_3 & \cdots & x_n & 0 & 0 & \cdots & 0 \\ 0 & 0 & x_1 & x_2 & x_3 & \cdots & x_n & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & x_1 & \cdots & x_{n-2} & x_{n-1} & x_n & 0 \\ 0 & \cdots & 0 & 0 & 0 & x_1 & \cdots & x_{n-2} & x_{n-1} & x_n \end{bmatrix}$$

Figure 1. Toeplitz Matrix

computing $M_{1,a}M_{2,a}...M_{L,a}XM_{L,b}^T...M_{2,b}^TM_{1,b}^T$. Note that during training, we must compute this manually, but during testing, we freeze our weights, so we can just precompute $M_{T,a} = M_{1,a}M_{2,a}...M_{L,a}$ and $M_{T,b}^T = M_{L,b}^T...M_{2,b}^TM_{1,b}^T$ to get the result of our convolution as $M_{T,a}XM_{T,b}^T$. Note that the related work contains the full proof as to why this process works. This describes the algorithm used to both initialize our Low-Rank Convolution Layer and to apply our Low-Rank Convolution onto an image X. During training, we simply pass M_a, M_b as parameters into PyTorch's optimizer and utilize PyTorch's in-built autogradients/backward.

5. Architecture Methodology

Now that we have described the algorithmic manner by which our Low-Rank Convolutional Layer works, we now describe the architecture that we use throughout our experiments and project. For our architecture, we create a mini version of the Very Deep Convolutional Network by VGG [8]. This architecture stacks together a large number of convolutional blocks and has shown to obtain very high performance.

Note that a convolutional block ConvBlock(C_1, C_2) consists of following layers in the following manner:

1. Conv Layer ((In,Out) Channels: (C_1, C_2)) + ReLU
2. Conv Layer ((In,Out) Channels: (C_2, C_2)) + ReLU
3. Max Pooling 2D (Stride: 2, Window Size: $(2, 2)$)

The idea is that each block will half the width and height of each image. (due to the Max Pool layer). To compensate, we double the channels during each convolution block, to have the total features be halved when moving from one convolution block to the next. For our experiments, we standardized each convolutional layer in a block to have the same kernel size (dependent on the experiment) and use "same" padding. For our architecture, we simply stack these ConvBlocks together, ending with linear layers:

1. ConvBlock(3, 32)
2. ConvBlock(32, 64)
3. ConvBlock(64, 128)
4. Flatten + Linear($2048 \rightarrow 128$) + ReLU
5. Linear($128 \rightarrow 10$) + Softmax

A visual rep. of the architecture can be seen in Figure 2.

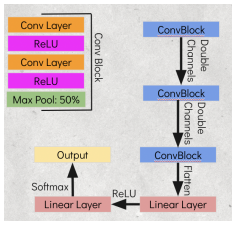

Figure 2. Model Architecture

6. Experiments

We ran many experiments and collected a large amount of data from each experiment during our project. We first state that we ran four large scoped experiments. These were:

1. Baseline Architecture
2. BatchNorm Architecture (Insert a 2d BatchNorm layer after each Convolutional **Layer**)
3. Dropout Architecture (Insert a $p = 0.4$ Dropout layer after each Convolutional **Block**)
4. Both Architecture (Utilize BatchNorm and Dropout)

For each of these larger scoped experimenst, we ran experiments using kernel sizes of $K \in \{3, 5, 7\}$. For each kernel size K, we ran the experiment using rank $L \in \{1, 2, 3, ...K\}$. Further, for each experiment with a architecture type, kernel size K, and rank L, we reran the entire experiment 50 times using 50 random seeds. This was to ensure that our average times were as accurate as possible by collecting a sizable sample. Note that during each experiment we used a 5000/1000 train/test split for each of the 10 image classes, batch learning with a batch size of 64, 10000 training iterations, cross entropy loss, an Adam Optimizer, tested on the entire 10000 image test set, and logged important information.

7. Results

When examining the results, we can first examine a comparison of results between the different types of architectures at rank-1 and kernel size of 5 and 7. We first note that we initially ran our experiments only with the Base model (without dropout or BatchNorm), but noticed that the model was overfitting. In order to combat this, we experimented with these techniques,

regularly utilized to combat overfitting, and in the process were able to test how low-rank convolutions would work when additional layers were added to the model.

Arch Type	Test Loss	Test Acc	Test Time	Train Loss	Train Acc	Train Time
Base (7)	1.046	72.67	1.398	0.27	90.59	138.36
BatchNorm (7)	0.796	78.52	6.796	0.283	90.47	147.31
Dropout (7)	0.768	74.32	1.602	0.569	79.78	140.15
Both (7)	0.669	78.09	7.103	0.506	82.48	148.12
Base (5)	1.062	75.5	1.136	0.159	94.62	91.40
BatchNorm (5)	0.767	78.78	4.564	0.262	90.99	99.99
Dropout (5)	0.701	76.41	1.336	0.533	81.03	92.72
Both (5)	0.662	78.08	4.605	0.524	81.86	101.39

Figure 3. Table of Results for Architecture Types

We can generally observe that, firstly, BatchNorm seems to be consistently more computationally expensive as each of the architectures containing BatchNorm took longer during both training and testing than their respective counterpart and at approximately the same rate longer. This begins to display that the convolutional layer might not be the bottlenecking component in the time it takes the model, but this is further exlpored later. We can also observe that BatchNorm seems to create the most generalizable models. At both kernel sizes of 5 and 7, models with BatchNorm had test accuracies much closer to their respective training accuracies, indicating better generalizability and less overfitting. Further, the architecture which seemed to have the best generalizability and test loss was the one using both Dropout and BatchNorm. We note that at kernel sizes of both 5 and 7, this architecture had the best test loss and nearly the best testing accuracy as well, despite having a training accuracy which was nearly 10% under the best. The drop in training accuracy is acceptable because it simply displays that the model has not reached its full potential yet, meaning that this architecture is conducive to good performance. Throughout the rest of this paper we examine the results from the architecture using both BatchNorm and Dropout.

7.1. Timing Results

Figures 4 and 5 depict the results that we had from low-rank convolutions when evaluating the time taken to test and the time taken to train the model. Note that at $x = 1.0$ (rightmost point on the plots) the $\frac{Rank}{Kernel\_Size}$ value is 1.0, meaning that our kernel is full-rank and we actually just use PyTorch's Default Convolution. We use $x = 0.0$ (leftmost point on the plotos) to represent rank-1 representations.

We can first observe that the time taken to test is essentially identical throughout all ranks at the same kernel size K. The differences are fractions of a second when testing 10000 images, which is negligible. This is relatively positive as it indicates that our low-rank convolutions are actually equivalent to

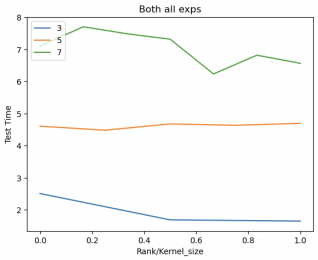

Figure 4. Time to Test

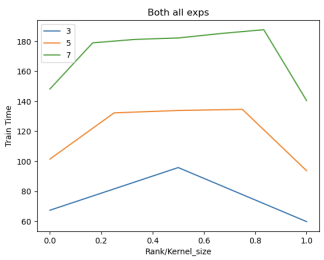

Figure 5. Time to Train

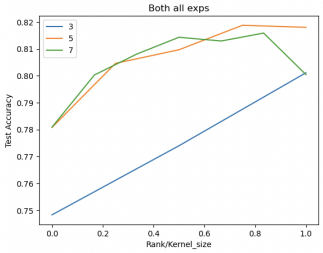

Figure 6. Test Accuracy

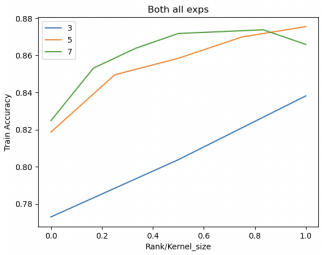

Figure 7. Train Accuracy

PyTorch's default convolution. However, we will also note that because we can always just precompute our convolution matrix and input the result into PyTorch's convolution function, we can always guarantee (in an actual setting) that using low-rank convolutions can always achieve the same efficiency as PyTorch. However, despite our implementation not having all the same under-the-hood optimizations as PyTorch, we obtain the same efficiency, indicating that with more optimizations under-the-hood, our low-rank convolutions can easily outperform PyTorch. However, we note that these results in combination with our results from BatchNorm experiments lead to the conclusion that the convolutional layer is not the time-bottleneck for our model, and instead other layers like BatchNorm are.

We now analyze training time, which was a concern listed in related works about our project idea. Firstly, we can observe that as the rank of the kernel increases, the time taken to train increases as well. We notice a sharp increase when moving from rank-1 to rank-2, which emphasizes the importance of rank-1 (highly parameter efficient) representations. In fact, while PyTorch's Default Convolution (at $x = 1.0$) barely trains faster than our Rank-1 model, our Rank-1 model remains highly competitive in total train time. This is a highly important result as it shows that we are able to mitigate the training time concerns of constraining training to a low-rank space for all of training.

7.2. Performance Results

We can examine Figures 6 and 7 to identify our low-rank convolutions' performance on the testing set and training set (after training). Firstly, we observe that the accuracy during both training and testing generally increases as the rank that we use to represent the kernel increases. This is expected because with more parameters, we would expect greater performance. However, we note that in testing accuracy, our Rank-1 model is actually only approximately 2% off in accuracy from PyTorch's Default Convolution, which is a relatively minor accuracy dropoff. This is a very promising result as it displays cutting down parameters from $O(K^2)$ to $O(K)$ can almost entirely maintain accuracy. Another important and positive result is that the training accuracy increases more quickly (as rank increase) when compared to testing accuracy. This indicates that lower rank models are more generalizable and less prone to overfitting as the difference between train and test performance is minimized at lower ranks. Again, this also makes sense as reducing parameter counts in models generally helps to reduce overfitting as it forces the model to learn more generalizable

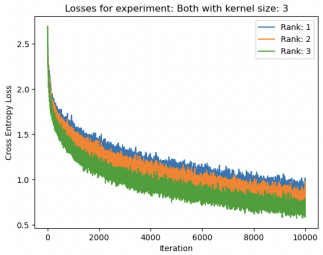

Figure 8. Loss Kernel Size 3

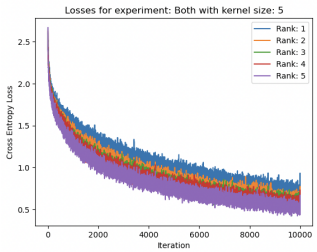

Figure 9. Loss Kernel Size 5

schemes. Another interesting result was that our Rank-6 model for a kernel size of 7 actually outperformed PyTorch's Default kernel size 7 (full-rank). We explore this result in more detail.

7.3. Performance over Time Results

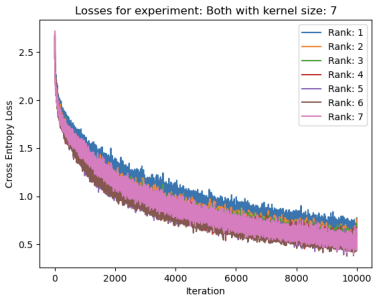

Figure 10. Loss Kernel Size 7

For this portion, we examine Figures 8, 9, and 10. More specifically, we look at iteration 2000 in each of these figures to evaluate how the performance of the model is early into training. This helps us evaluate how quickly the model can reach good performance and how quickly it trains (NOT related to time taken to train). We first observe that at 2000 iterations, the difference in performance between rank 2 and rank 3 for a kernel size of 3 is fairly large, the difference in performance between rank 4 and rank 5 for a kernel size of 5 is smaller, and rank 6 outperforms rank 7 for a kernel size of 7. The general observable trend is that as the kernel size increases, low-rank convolutions obtain performance comparatively more quickly than PyTorch's Default Convolutions. This is important for our project as it displays that even though low-rank convolutions may take more time for the same number of iterations, depending on the task, it is possible to simply train them for less iterations and achieve the same performance, thereby limiting training time as well. This is a good depiction that our low-rank convolutions are actually going to scale better than the default convolution. With larger models and larger kernels, we would then expect that our low-rank convolution to see less drawbacks in terms of train time and time needed to obtain good performance due to this improved scaling.

8. Conclusion

Overall, our project was a success. We firstly displayed how we could cut use low-rank matrices to reduce memory in a guaranteed fashion, cut the number of parameters used by convolutions in CNNs from $O(K^2)$ to $O(K)$, and how we could constrain weights to these low-rank spaces throughout the entirety of training. We then discovered that, even with an unoptimized under-the–hood implementation, our low-rank convolutions were essentially identical to PyTorch's Default Convolutions in testing efficiency (time taken to evaluate on testing set). Unfortunately, our low-rank convolution, even at Rank-1, was a bit more inefficient during train time when compared to PyTorch's default convolution. However, there are 3 main caveats to this result. Firstly, we displayed in section 7.3 that low-rank convolutions scale better, meaning that they often achieve better performance quicker, minimizing the number of iterations needed. This could enable reducing the number of training iterations, thereby potentially reducing the impact of this longer training time. Secondly, we observed that other layers in the model (such as BatchNorm) often had a much larger impact on time taken than the convolutional layers, meaning that convolutions alone are not the bottleneck for timing. Finally, we note that, in practice, testing time is often significantly more important than training time, meaning that this tradeoff is likely worth it, especially considering the memory improvements our approach guarantees.

In addition to the time taken, we showed that the accuracy dropoff when using low-rank and even rank-1 convolutions was actually relatively negligible. With only a 2% dropoff in testing between a rank-1 convolution and PyTorch's convolution, we conclude that limiting the number of parameters can be done in this form without have a large impact on model performance. Further, we show that the low-rank convolutions are actually more generalizable and less prone to overfitting, making them highly usable in settings with limited data (likely to overfit). Finally, we also display that low-rank convolutions do not need to be standalone as performance of the models was actually improved when these low-rank convolutions were combined with other layers such as BatchNorm and Dropout, and these actually helped increase generalizability even further. Overall, our project displays a time-efficient method to reduce the memory from order $O(K^2)$ to $O(K)$ while maintaining accuracy and increasing generalizability.

When examining future work, there are 2 components we believe should be directly worked on. Firstly, a more optimized under-the-hood implementation for low-rank convolutions and multiplications should be applied so that train time can be even faster, potentially rivaling PyTorch, and test time can be faster as well, potentially beating PyTorch. In addition, another area of future work would be to explore the same low-rank weight constraint throughout the entirety of training to larger CNNs and other large models to see how efficiency gains might be achieved there.

9. Individual Contributions

Each member of the team contributed equally to the project. We worked through the code together and were always together when we completed the code. We ran all the experiments at once on a PC with a GPU. We also equally worked on the paper and presentation together while in the same room.

References

[1] Cifar-10 and cifar-100 datasets. 1

[2] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, Sep 1936. 2

[3] Robert M Gray et al. Toeplitz and circulant matrices: A review. *Foundations and Trends® in Communications and Information Theory*, 2(3):155–239, 2006. 1

[4] Soufiane Hayou, Nikhil Ghosh, and Bin Yu. Lora+: Efficient low rank adaptation of large models, 2024. 1

[5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. 1

[6] Sri Jaladi. Using rank-n approximation and matrix multiplication to speed up convolutions in convolutional neural networks (cnns), Dec 2022. 1

[7] Ziran Qin, Mingbao Lin, and Weiyao Lin. Low-rank winograd transformation for 3d convolutional neural networks. 2023. 1

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. 2

[9] Yang Yang, Wen Wang, Liang Peng, Chaotian Song, Yao Chen, Hengjia Li, Xiaolong Yang, Qinglin Lu, Deng Cai, Boxi Wu, and Wei Liu. Lora-composer: Leveraging low-rank adaptation for multi-concept customization in training-free diffusion models, 2024. 1