FAST COMPUTATION OF WIDE NEURAL NETWORKS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Vineeth Chigarangappa Rangadhamappa

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Chigarangappa Rangadhamappa, Vineeth M.S., Purdue University, December 2018. Fast computation of wide neural networks. Major Professor: Vaneet Aggarwal .

The recent advances in artificial neural networks have demonstrated competitive performance of deep neural networks (and it is comparable with humans) on tasks like image classification, natural language processing and time series classification. These large scale networks pose an enormous computational challenge, especially in resource constrained devices. The current work proposes a targeted-rank based framework for accelerated computation of wide neural networks. It investigates the problem of rank-selection for tensor ring nets to achieve optimal network compression. When applied to a state of the art wide residual network, namely WideResnet, the framework yielded a significant reduction in computational time. The optimally compressed non-parallel WideResnet is faster to compute on a CPU by almost 2x with only 5% degradation in accuracy when compared to a non-parallel implementation of uncompressed WideResnet.

1. INTRODUCTION

The recent advances in artificial neural networks (ANNs) have demonstrated competitive performance of deep neural networks (and it is comparable with humans) on certain tasks like image classification [1], natural language processing ([2], [3]) and time series classification [4]. With each advance, these networks have grown larger and deeper. Accordingly, the number of trainable neural network parameters have increased steeply, which in-turn led to an increased computational time to train a neural network. For instance, the pioneering convolution neural network (CNN) architecture for image recognition, namely LeNet [5], had 0.04 million trainable parameters, while the state-of-the-art CNN architectures, such as ResNet-50 [6] and NASNet [1] possess 25 million and 84.7 million trainable parameters respectively.

Although large number of network parameters significantly improve the predictive performance of a neural network there exists a demonstrative redundancy [7] in the parameterization of many deep learning models. This leads to model over-fitting and hence poorer generalizability. Further, large networks pose significant computational challenge for resource-constrained embedded devices like smart-phones and Internet-of-things devices. Hence, there is a necessity to reduce energy requirements i.e., the computational time of the ever-growing ANN architectures.

Convolution is an expensive operation that contributes to the bulk of the computation time in the state-of-the-art neural nets, especially wide nets. Computational time of convolution operation can be reduced either by (i) reducing the layer complexity i.e., reducing the number of network parameters or (ii) by implementing the convolution operation in a parallel computing environment. There exist multiple approaches to reduce layer complexity, such as network pruning [8], sparsity regularization [9], and low rank approximation [10]. Similarly there exist several parallelism strategies for implementing convolution neural networks such as intra-layer parallelism [11], model parallelism and data parallelism [12].

There exists a framework, namely Wide-Compression [13], that significantly reduces layer complexity by compressing over-parameterized filter channels with minimal loss in accuracy. But, this framework does not compress the network optimally and its real impact on the gain in computational speed-up is not investigated. Further, the framework also possess a novel within-layer (sub-layer) parallelism that can be applied in addition to the afore-mentioned parallelism techniques.

The current work proposes a methodology to compress a wide network optimally by means of a targeted rank selection for individual network layers. In addition, the current work also demonstrates the within-layer (sub-layer) parallelism for maximal reduction in a wide neural layer's computational time. The significant compute-time reduction is demonstrated through an application to WideResnet architecture. In the end, we discuss a couple of empirical scenarios that help in showcasing the capability of the framework and conclude with a set of future research ideas.

The rest of the report is organized as follows: a description of the tensor basics, Tensor Ring Nets compression framework and current research methodology is provided in chapter 2. Then, an overview of the benchmark dataset and the network architecture is provided in chapter 3. Subsequently, I present different empirical scenarios and the associated results in chapter 4. Finally, chapter 5 provides a conclusion and chapter 6 lists some ideas for the future work.

2. METHODOLOGY

For convolution neural networks, the convolution operation in a layer with an input $\mathcal{X}$, kernel $\mathcal{K}$ and output $\mathcal{Y}$, is given by the following equation:

$$\mathcal{Y}_{h,w,o} = \sum_{d_1=1}^{D} \sum_{d_2=1}^{D} \sum_{i=1}^{I} \mathcal{X}_{h',w',i} \mathcal{K}_{d_1,d_2,i,o} \tag{2.1}$$

where

$$\text{h'} = (\text{h-1})\text{s}+d_1\text{-p}$$
$$\text{w'} = (\text{w-1})\text{s}+d_2\text{-p}$$

Here, s is the stride, p is the zero-padding size, I is the number of input channels and O is the number of output channels. Further, h and w represent the height and width dimensions of the input. The bias term is omitted for clearer representation.

To determine the computational time of the above convolution operation, the number of primitive operations such as multiplication and addition, have to be determined. In a computational environment the numbers are represented by floating points and hence the primitive mathematical operations are called floating point operations, shortly flops.

It has to be noticed that input $\mathcal{X}$, kernel $\mathcal{K}$ and output $\mathcal{Y}$, are tensors. In the following section, a brief introduction to tensors, tensor operations and their accompanying flops are provided.

2.1 Tensor Basics

Tensors are multidimensional arrays. For example, a matrix is a tensor of 2^{nd} order. Similarly, a vector is a tensor of 1^{st} order and a scalar is a tensor of 0^{th} order.

Every element in a 2nd order tensor i.e., matrix, can be represented by A_{ij}. Further, when two matrices of sizes $m \times p$ and $p \times n$ are multiplied then an element in the resultant matrix can be represented by the following equation:

$$A_{ij} = \sum_{k=1}^{p} B_{ik} C_{kj} \tag{2.2}$$

Thus, to obtain a single element in the resultant matrix A, one has to perform 'p' multiplications. Since there exist *mn* elements in the resultant matrix A, the total number of multiplicative operations will be *mnp*.

Flop is an acronym for 'floating point operation'. The number of floating point operations, shortly flops, is the traditional measure of efficiency of a numerical algorithm. There do not exist a standard definition of flops. It may include the count of any one floating point operation (like an addition or a multiplication) or it may include the count of both additive and multiplicative operations. We adopt the former definition through out this report. Hence the number of flops required to obtain the resultant matrix A in equation 2.2 is *mnp*.

Equation 2.2 also described an *index contraction* event. Notice that the index 'k' was contracted by summing over all the possible 'p' values. Similarly in higher order tensor multiplication, either one or multiple indices can be contracted. For example:

$$A_{ijlm} = \sum_{k=1}^{p} B_{ijk} C_{klm}$$

Tensor multiplication can be easily visualized using diagrams. Figure 2.1 shows tensor diagrams for scalar, vector, matrix and order-3 tensor [14].

The circles represent tensors while the edges emanating from circles indicate their axes. Similarly, a product between two tensors can be represented using a chain of tensors as shown in Figure 2.2.

Figure 2.2 shows a matrix product with the contracted axes dimension 'k' in the first step and a product of two order-3 tensors with the contracted axes dimension 'r' in the second step. The flops of tensor product can be computed by taking the product

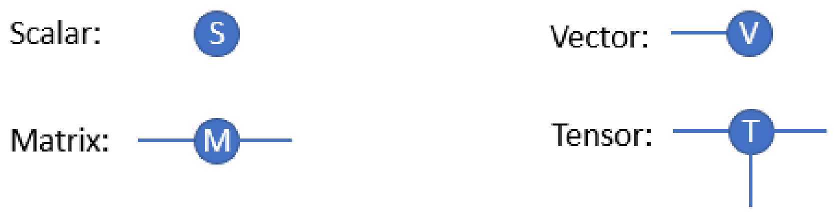


Figure 2.1. Diagrammatic notation for tensors

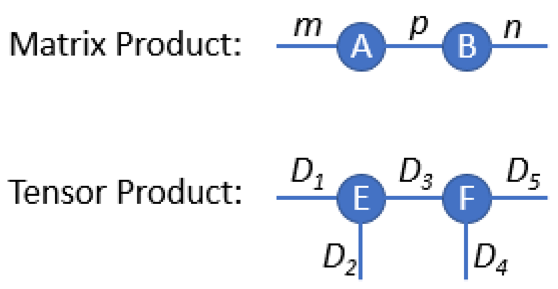


Figure 2.2. Diagrammatic notation for tensor products

of dimensions with the contracted dimension appearing only once. For example, in Figure 2.2, the matrix product requires mnp flops while the tensor product requires $D_1D_2D_3D_4D_5$ flops.

If two or more tensors are connected with a common contracted axes then that set-up is called a tensor network (TN). An important property of TN is that the order of merging two tensors determines the computational efficiency of merging. Figure 2.3 shows product of 3 tensors with two different merge orders [14]. The number of operations required to obtain the same final result of a TN contraction is of the order of D^4 for first case while it is of the order of D^5 for the second case, where D is each axis dimension. Hence, one has to carefully choose the merge order for optimal computational time.

For a detailed theoretical description on tensors, the reader is redirected to [15] and [14].

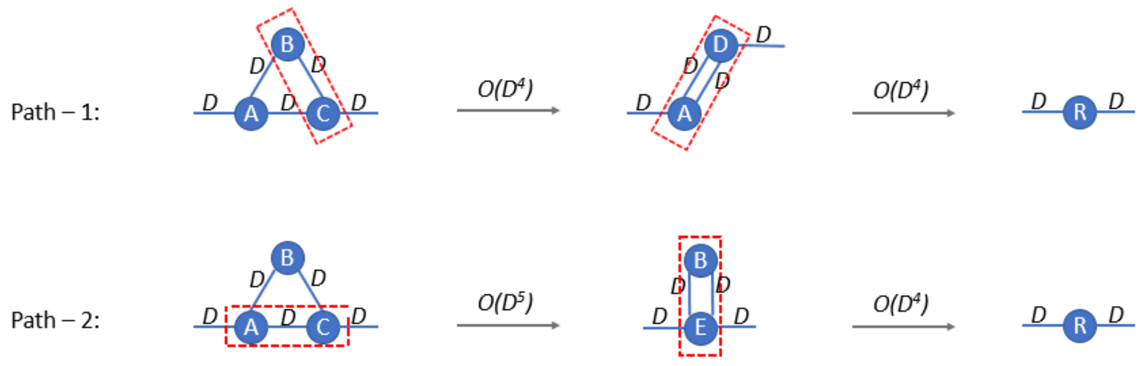


Figure 2.3. A TN contraction that shows different merge orders and accompanied flops

2.2 Tensor Factorization

The convolution operation contributes to the majority of computational time of a neural network layer and it is directly dependent on the layer size. The kernel tensor determines the layer size. Therefore, to reduce the computational intensity of convolution, a layer's kernel tensor size ought to be reduced.

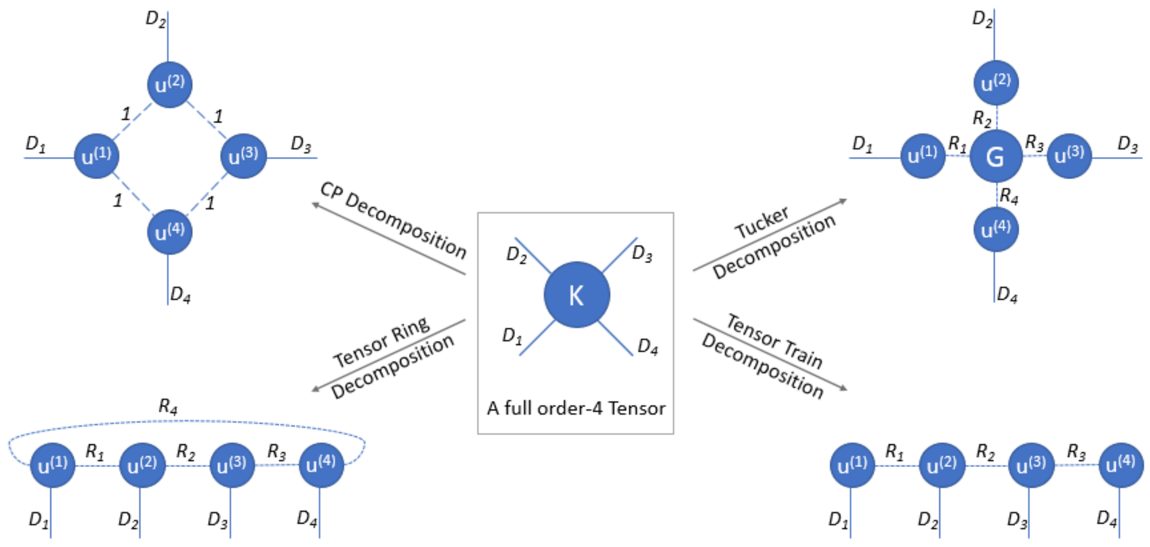


Figure 2.4. Four popular tensor decompositions to obtain lower order tensor factors

Just as a large matrix can be approximated with two lower rank factor matrices by means of an appropriate low rank matrix factorization like singular value decomposition, there exist four popular tensor decomposition methods that reduce the size of higher order tensors. They are:

1. Canonical decomposition (CP decomposition)
2. Tucker tensor decomposition
3. Tensor train decomposition
4. Tensor ring decomposition

Figure 2.4 shows a sample decomposition of a large tensor into 6 factored tensors of order-3. Among the four popular tensor factorization methods, although tucker factorization [16] is good for compressing neural network layers [17], it was shown that tensor ring decomposition has higher expressive power in data representation [18] for the same intermediate rank and so they were found to be better for compressing convolution layers in neural network [13]. Hence Tensor Ring Network (TRN) based decomposition was chosen for the current implementation.

2.3 Prior Work: Tensor Ring Network (TRN) Compression

It can be observed that the kernel tensor $\mathcal{K}$ as described in equation 2.1 is a 4^{th} order tensor. In the TRN decomposition, the 4^{th} order tensor is decomposed into four order-3 tensors of rank 'R'. Further, to retain the spatial characteristics of the tensor for convolution, two order-3 tensors were merged to retain the spatial dimension of $\mathcal{K}$, resulting in a single order-4 tensor. Thus, the kernel decomposition is as follows:

$$\mathcal{K}_{d_1,d_2,i,o} = \sum_{r_1=1}^{R}\sum_{r_2=1}^{R}\sum_{r_3=1}^{R} u^{(2)}_{r_3,I,r_2} u^{(1)}_{r_2,d_1,d_2,r_1} u^{(3)}_{r_1,O,r_3} \tag{2.3}$$

Further, for large number of input channels 'I' and output channels 'O', as is the current case, $u^{(2)}$ and $u^{(3)}$ are further factorized into three smaller order-3 tensors as follows:

$$u^{(2)}_{r_3,I,r_2} = \sum_{r_4=1}^{R}\sum_{r_5=1}^{R} \hat{u}^{(1)}_{r_3,I_1,r_4}\hat{u}^{(2)}_{r_4,I_2,r_5}\hat{u}^{(2)}_{r_5,I_3,r_2} \tag{2.4}$$

$$u^{(3)}_{r_1,O,r_3} = \sum_{r_6=1}^{R}\sum_{r_7=1}^{R} \hat{u}^{(4)}_{r_3,O_1,r_6}\hat{u}^{(5)}_{r_6,O_2,r_7}\hat{u}^{(6)}_{r_7,O_3,r_1} \tag{2.5}$$

where I = $I_1I_2I_3$ and O = $O_1O_2O_3$.

The kernel factorization 2.3 is integrated into the original convolution operation 2.1 to obtain TRN based convolution. This TRN compression based convolution is executed in 3 steps as follows [13]:

step 1

$$\mathcal{P}_{h',w',r_2,r_3} = \sum_{i=1}^{I} \mathcal{X}_{h',w',i}u^{(2)}_{r_3,i,r_2} \tag{2.6}$$

step 2

$$\mathcal{Q}_{h,w,r_2,r_3} = \sum_{d_1,d_2=1}^{D}\sum_{r_2=1}^{R} \mathcal{P}_{h',w',r_2,r_3}u^{(1)}_{r_2,d_1,d_2,r_1} \tag{2.7}$$

step 3

$$\mathcal{Z}_{h',w',O} = \sum_{r_1,r_3=1}^{R} \mathcal{Q}_{h,w,r_2,r_3}u^{(3)}_{r_1,O,r_3} \tag{2.8}$$

The layer output $\mathcal{Z}_{h',w',O}$ is obtained by performing three steps wherein each step is a convolution step. Step 1 and step 3 can be recognized as a 1x1 convolution as $u^{(2)}_{r_3,I,r_2}$ and $u^{(3)}_{r_1,O,r_3}$ can be reshaped into a higher order tensors $u^{(2)}_{1,1,r_3,I,r_2}$ and $u^{(3)}_{1,1,r_1,O,r_3}$ respectively. Thus each of those two steps represent r_3 convolutions with filter size

1x1. While the step 2 can be viewed as a convolution with filter size DxD and with input filters r_2 and output filters r_1.

The TRN network that results from the above scheme for a very wide layer, with large I and large O, is shown in the figure 2.5. The dashed lines in the figure indicate the convolution operation as described by equation 2.7. While contracting this TRN network, care has to be taken when merging the input with all the factored tensors.

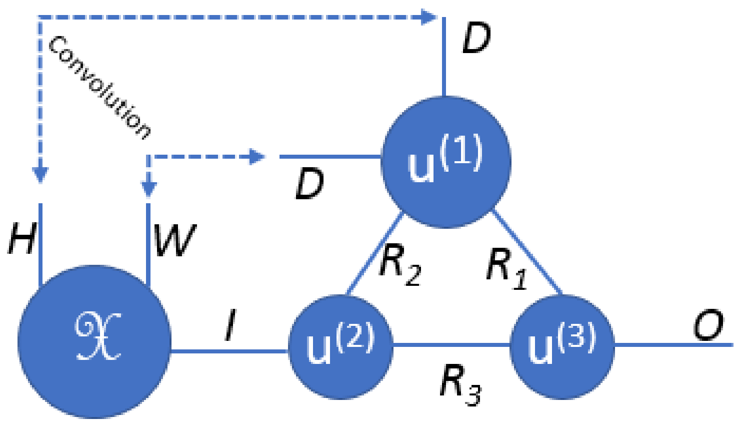


Figure 2.5. Tensor network diagram for a layer's TRN based convolution

Finally, we compare the number of flops required to execute a regular convolution as described in equation 2.1 and the 3-step TRN based convolution. In general the number of flops for a convolution operation as described by equation 2.1 is given by HWD^2IO and when a mini-batch is used instead of a single input image, the regular convolution flops are given by the equation:

$$flops_{Reg} = BHWD^2IO \tag{2.9}$$

Similarly, flops for each of the 3steps in TRN based convolution [equations 2.6, 2.7, 2.8] is computed as follows:

- Step 1: It is a convolution with 'I' input channels and 'R^2' output channels. Therefore number of flops are HWR^2I.

- Step 2: It consists of 'R' convolution with 'R' input channels and 'R' output channels. Therefore number of flops are HWR^3D^2.
- Step 3: It is a convolution with 'R^2' input channels and 'O' output channels. Therefore number of flops are HWR^2O.

Thus, the total number of flops for a TRN based convolution with a batch size 'B', is given by the equation:

$$flops_{TRN} = B(HWR^2I + HWR^3D^2 + HWR^2O) + 4R^3(I+O) \tag{2.10}$$

where, $4R^3(I+O)$ represent the upper bound for flops required to obtain $u^{(2)}$ and $u^{(3)}$ as described in equation 2.5

Finally, the reduction in computational complexity with TRN based convolution when compared with regular convolution can be quantified by the parameter C_{conv} [13], given by the following equation:

$$C_{conv} = \frac{BIOD^2}{4R^3(I+O) + BR^2(I+O) + BR^3D^2} \tag{2.11}$$

2.4 Original Contribution: Targeted Rank-Selection Framework

The prior work, as described in the earlier section and herein referred to as TRN framework, demonstrated that the network can be compressed by significant amount for different factored tensor ranks. In doing so, a single rank was chosen for all factored tensors in different layers of the network. But, as the number of filters vary in each layer and the TRN framework only compresses along the filter dimensions it is not well motivated to chose a single rank for all layers from the computational point of view. Although, a single-rank choice results in lower number of network parameters, it does not provide optimal reduction in the network's computational time.

Moreover, it was noticed that the naive implementation of TRN based compressed network was computationally slower when compared with uncompressed WideResnet.

I surmise that the problem arose because of a naive implementation that did not account for the parallelism of deep-learning computational library. Further, I explore the parallel implementation of TRN framework at a layer level and quantify the actual reduction in runtimes at the layer-level.

In summary, overcoming the computational environment issues and performing the targeted rank-selection form the basis for the current work. Hence, the current work can be regarded as an improvement in the practical implementation of theoretical work as described in section 2.3. An overview of the research methodology, followed in the current work, is provided in table 2.1.

Table 2.1.
Summary of the research methodology followed in the current work

Research Methodology	
Step 1:	Profile the single-rank compressed WideResnet network to identify the sub-optimally compressed layers w.r.t runtime
Step 2:	Identify and resolve the computational environment issues that slowed down the naive implementation of compressed network
Step 3:	Perform targeted rank selection i.e., choose appropriate rank for individual layers so as to obtain optimal speed-up
Step 4:	Experiment with network architecture to obtain faster speeds for similar accuracy when compared to the baseline network

The following sub-sections describe each of the steps in detail, including but not limited to the computational tools that were used to complete the analysis.

All the artificial neural networks trained in the current project were implemented in Python. There exist different open-source computational libraries for training artificial neural networks like Caffe, pyTorch, Tensorflow and Theano. Among them, Tensorflow library [19] has been chosen for its ease of implementation, availability of fairly detailed documentation, state-of-the art research models and a thriving supporting community on 'stackoverflow.com' [20].

Tensorflow builds a network graph and executes it in a dedicated session inside python environment. Tensorflow operations like matrix multiplication, application of non-linear activation function, batch normalization etc., form the nodes while tensors exchanged between the operations form the edges of the graph. An efficient placement of graph nodes in the execution pipeline is of paramount importance as we are concerned with the runtime of the network.

2.4.1 Identify bottleneck layers: Profiling network run-time

Since computation time of the neural network is of interest, it is imperative to obtain the computational time of the individual layers. Such an information about the runtime of each layer provides deeper insights that would enable a researcher to avoid computational bottlenecks in any layer and to tailor the compression level of each layer by means of optimum rank selection for an adequate balance between accuracy and run-time of the network.

Individual layer Run-time cannot be measured from python stack as Tensorflow executes the network graph inside a session. Therefore, an inbuilt profiling tool called 'tfprof', along with Tensorboard, is used to procure run-time information.

In Tensorflow, enabling the full code trace and passing it as an option to session run call gives us the runtime information. The Tensorflow session's summary file writer collects appropriate runtime meatadata information of the graph nodes into event files written by the summary file writer. The researcher should exercise caution when visualizing the runtime meta-data information in Tensorboard [version 1.5] as it displays cumulative time which is the sum of the time taken by a node and cumulative time taken by all preceding nodes in the execution pipeline. In all of the current analysis, the event files were processed through a suitable JSON-script to obtain the layer run-times.

As an example, Figures 2.6 and 2.7 depict the profiling result of a WideResnet network trained on a CIFAR-100 dataset. These figures provide an overview of the layer run-times during network evaluation. It can be observed that first few layers have higher run-time. Higher runtime in the initial layers is due to image size while moderately high run-time in the final layers is due to higher number of filters even though the image size is reduced by factor of 3.

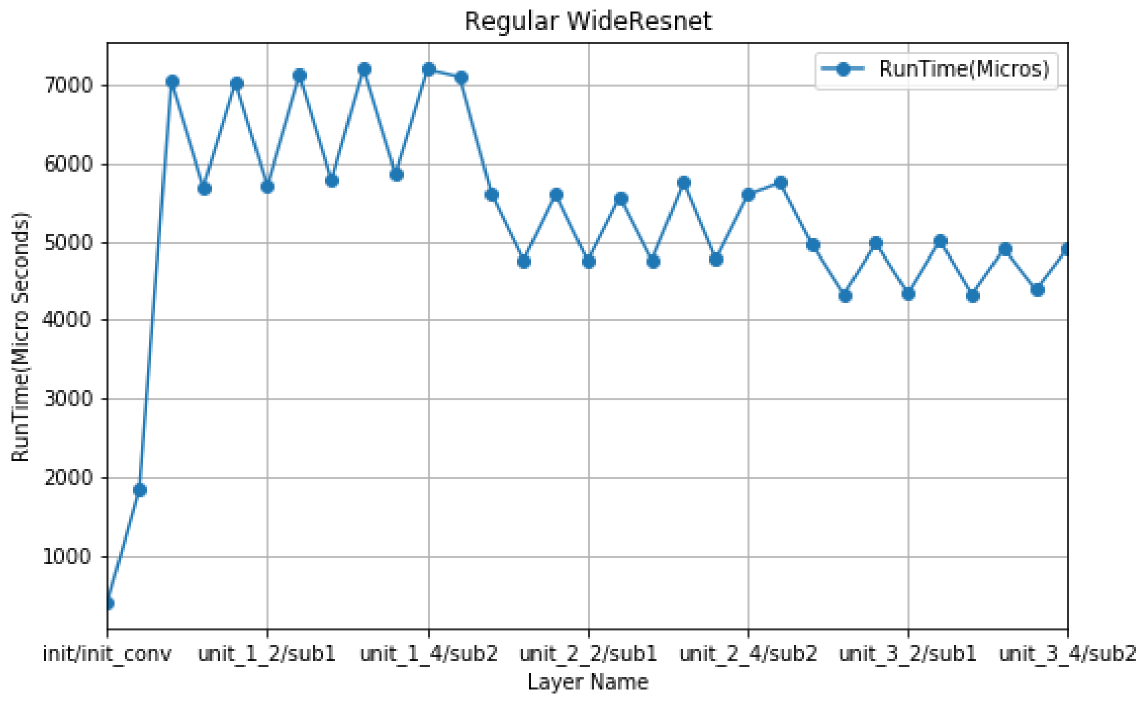


Figure 2.6. Layer-wise feed forward run-time in WideResnet

2.4.2 Computational environment issues: Tensorflow environment

As stated earlier, a naive implementation of TRN framework was found to be computationally slower. The main reason was the inefficient network graph execution in a tremendously parallel computing environment of a GPU.

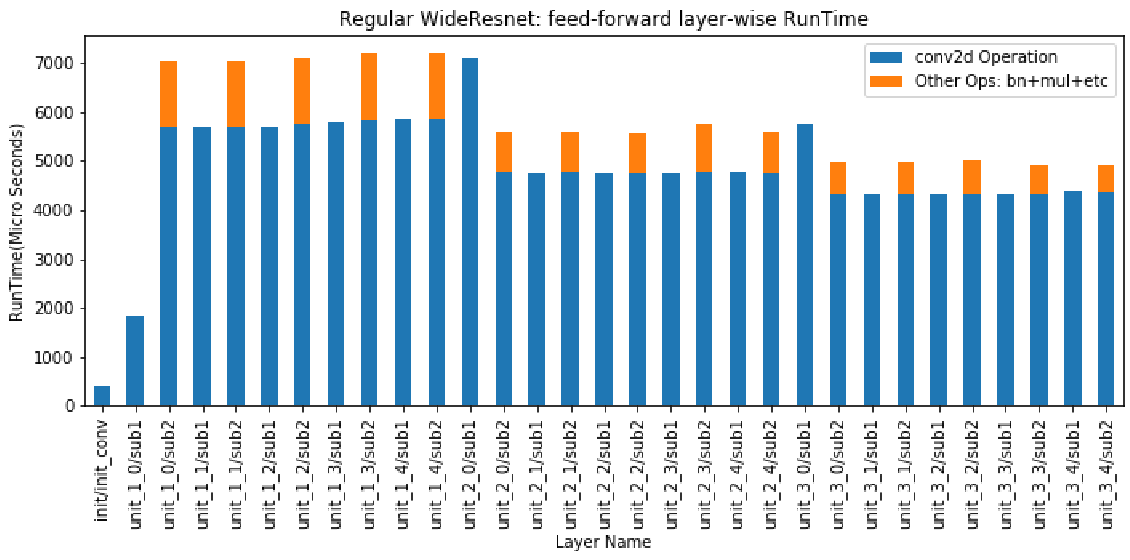


Figure 2.7. Proportion of convolution operation run-time of each layer in WideResnet

While executing the network graph on a GPU, Tensorflow maintains two GPU streams. In simple terms, the streams refer to the number of parallel computing threads that are being executed on a GPU. The two GPU streams are:

1. Compute-stream - It consists of all GPU operation kernels that perform mathematical computation.

2. Memory-copy stream - It consists of specific GPU kernels that are utilized for communicating (i.e., copying variables) to and from GPU and the host device (i.e., CPU).

Further, a GPU operation kernel may use multiple streams for computation while maintaining a single stream semantics. Each operation kernel of a GPU, when large enough, utilizes almost all the streaming multiprocessors (SM) in a GPU. At this stage, it is worth mentioning that each neural layer's convolution is executed by one instance of a corresponding GPU convolution kernel. This kernel is immensely parallelized wherein the multiple image patches of each image in a mini-batch is convolved with multiple filters in a parallel processing environment. Depending on the input tensor dimensions and the filter tensor dimensions, different partition sizes are created and executed on many SMs concurrently, resulting in significant speed enhancements.

But, in TRN based convolution, each layer's convolution is split into 3 sub-convolutions and each of those three are executed by three different instances of the GPU kernel. Therefore, each kernel instance receives thinner input and filter tensors which in-turn mean that the possible partition sizes of the underlying algorithm are smaller. In addition, the convolution operation sequence is such that the three steps has to be computed sequentially, as the convolution output of a step becomes the input of the subsequent step and hence this controlled data dependency further constrains the parallelizing algorithm underlying the GPU operation Kernel. Hence, TRN based convolution was not as fast as regular convolution when implemented naively on GPU without accounting for its' parallelism.

In the current work, we profile the network runtime on CPU as the lower-level functions that control the GPU parallelism were inaccessible. Moreover, the Tensorflow design did not allow for multiple GPU compute-streams as the Tensorflow creator's did not observe significant gains on enabling multiple compute streams. But, multiple streams was an essential necessity for the current work as it includes a parallel implementation of TRN based convolution. Hence, a CPU based computation was chosen for measuring the network run-times.

Before we compare the run-time, a controlled Tensorflow environment has to be setup as Tensorflows operations like tf.matmul, tf.Conv2d etc are heavily parallelized. Two types of parallelism exist:

- Intra_op_parallelism: In this scenario, an operation, say tf.Conv2d, makes use of multiple CPU threads to execute a layers convolution operation.

- Inter_op_parallelism: In this scenario, if there exist multiple tf.Conv2d operations that are independent, then they will be executed concurrently on the available CPU cores

In our flops computation, we never account for intra_op_parallelism. Further, inter_op_parallelism has to be controlled so that it doesnt result in an inefficient network graph during implementation. Hence, control on parallelism (i.e., removing intra_op_parallelism entirely and then controlling inter_op_parallelism) is essential to test our hypothesis that compute-time of TRN based convolution is less than that of regular convolution. These configurations are set via the tf.ConfigProto and passed to tf.Session in the 'config' attribute.

2.4.3 Selection of layer-wise ranks

The primary task is to select an appropriate single rank for the compressed neural network by computing the flops required for every layer in the network using equation 2.10. It has to be made sure that the total flops of the compressed network equal the total flops of an uncompressed network.

Subsequently, the layer-wise feed-forward runtime of the network is obtained and the bottleneck layers are identified. Among these layers, a select few layers that are most relevant to compress further are chosen and their ranks are reduced. Evaluate the performance of the networks for all of the lower ranks.

An overview of the framework is provided in table 2.2.

2.4.4 Experiments

The aim is to obtain a compressed network architecture that attains the same accuracy as an uncompressed WideResnet but is faster to compute. Therefore, we investigate the following two empirical scenarios:

Table 2.2.
Heuristic framework for targeted rank-selection

Targeted Rank Selection	
Step 1:	Select a single rank for all the layers in the network such that the flops of a compressed network and uncompressed network are equal
Step 2:	Identify, for that single rank, the bottleneck layers and reduce the rank of the factored tensors for those layers.
Step 3:	Evaluate the performance of the network for all the lower ranks
Step 4:	Choose a suitable rank for individual layer based on the desired accuracy and acceptable runtime of the compressed network

- Layer-wise targeted rank-selection for a compressed WideResnet with the default architecture
- Extend the Compressed WideResnet architecture such that the depth of the network is increased by 2x

For the details regarding each of the above scenarios, the reader is requested to consult section 4.2.

2.4.5 TRN compression's parallel implementation

The elegant 3-step TRN based convolution schema [Figure 2.8] lends itself magnificently to a parallel implementation while coding the algorithm in a parallel computational environment. It can be observed that each of the 3 steps consist of r_3 independent convolutions. These independent convolutions can be executed in parallel and accordingly the flops are:

$$flops_{TRNparallel} = B(HWRI + HWR^2D^2 + HWRO) + 4R^3(I + O)$$

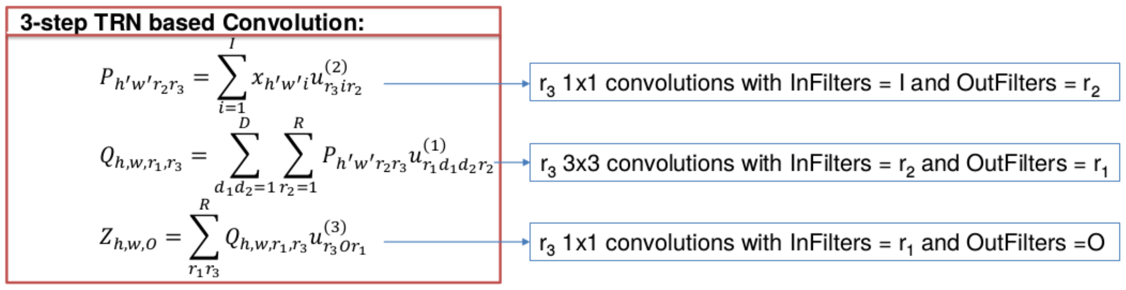

3-step TRN based Convolution:
$P_{h'w'r_2r_3} = \sum_{i=1}^{I} x_{h'w'i} u^{(2)}_{r_3 i r_2}$
$Q_{h,w,r_1,r_3} = \sum_{d_1 d_2=1}^{D} \sum_{r_2=1}^{R} P_{h'w'r_2r_3} u^{(1)}_{r_1 d_1 d_2 r_2}$
$Z_{h,w,O} = \sum_{r_1 r_3}^{R} Q_{h,w,r_1,r_3} u^{(3)}_{r_3 O r_1}$
r_3 1x1 convolutions with InFilters = I and OutFilters = r_2
r_3 3x3 convolutions with InFilters = r_2 and OutFilters = r_1
r_3 1x1 convolutions with InFilters = r_1 and OutFilters =O

Figure 2.8. TRN schema for parallelization

The TRN scheme, as shown in Figure 2.8, has to be implemented keeping in mind the ability to execute the 3 convolution sub-steps in parallel. In doing so, the prime concern is to make sure that the 3 convolution sub-steps are not in different parallel threads as it results in inefficient Tensorflow network graph. These three sub-steps must be grouped together and executed sequentially on one thread. On the other hand, there exist r_3 such groups and all of these r_3 groups can be executed in parallel as there doesn't exist any data-dependency between any two of the r_3 groups. In order to implement this parallel scheme, we take advantage of the Tensorflow's while loop implementation - tf.while, which makes it easy to organize the steps inside loop, if independent, to be executed in parallel.

3. DATASET AND NEURAL NETWORK ARCHITECTURE

3.1 CIFAR-100 Dataset

The CIFAR-100 dataset [21] is a collection of tiny images of size 32x32 with 50000 images in the training set and 10000 images in the test set. Every image is classified into one of the 100 classes. There exist 500 training images per class and 100 testing images per class. Figure 3.1 shows a few samples from the dataset.

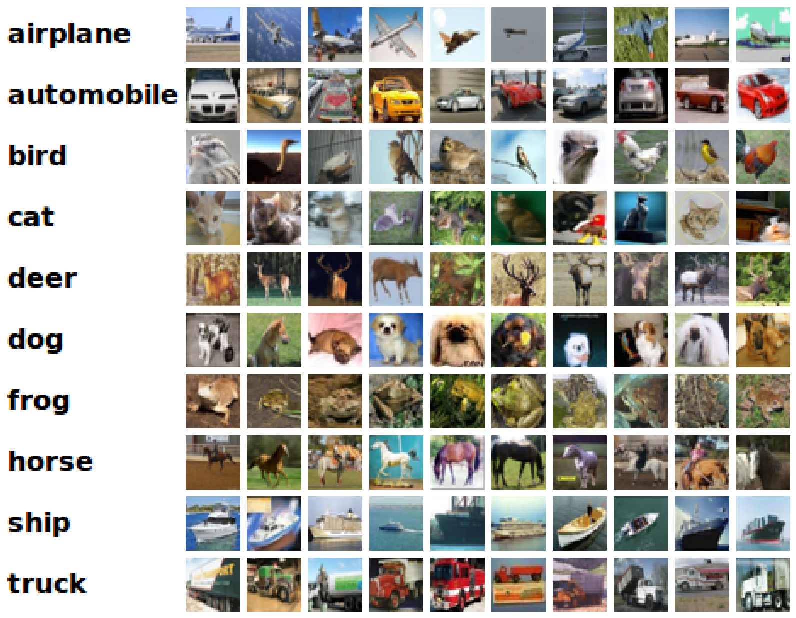


Figure 3.1. A sample of images and their class labels from CIFAR-100 dataset [21]

3.2 WideResnet Architecture

The state-of-the-art Resnet architecture ([6], [22]) was proposed to alleviate the problem of accuracy saturation in very deep networks. Thus, it enabled building deep and wide neural networks that provided state-of-the-art results on benchmark datasets including CIFAR-100, IMAGENET and COCO2015. Further studies on wide residual networks [23] showed that wide networks outperform deep networks. The width of the network represents the filter channel depth dimension and since we compress along the filter depth, it follows that the wider nets are more amenable to compression using the current TRN based compression scheme.

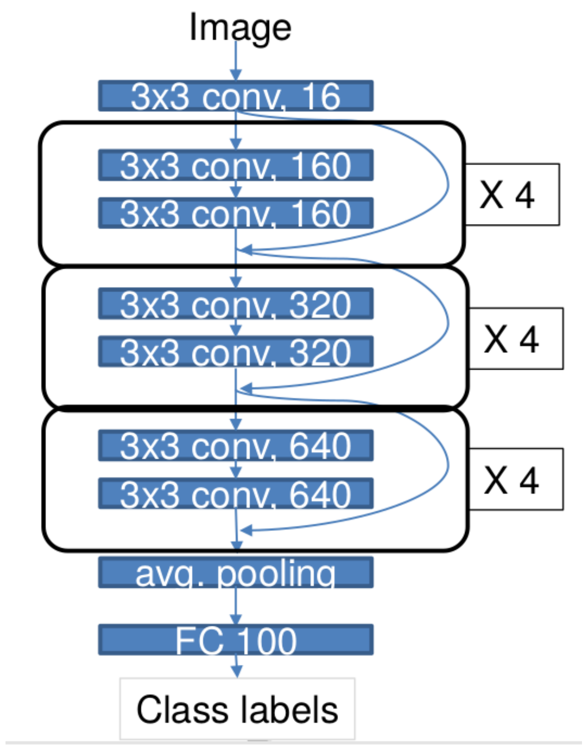


Figure 3.2. WideResnet Architecture

The WideResnet architecture consists of 28 layers with most of them being convolution layers, except for a fully connected layer and a terminal softmax layer. The convolution layers are segmented into 3 units such that each unit consists of 4 residual blocks. Every residual block consist of 2 convolution layers and a short-cut connection that bypasses those two layers. Before the start of each convolution layer, there exists a batch normalization layer whose output is fed to a non-linear activation function. An overview of the architecture is shown in the Figure 3.2.

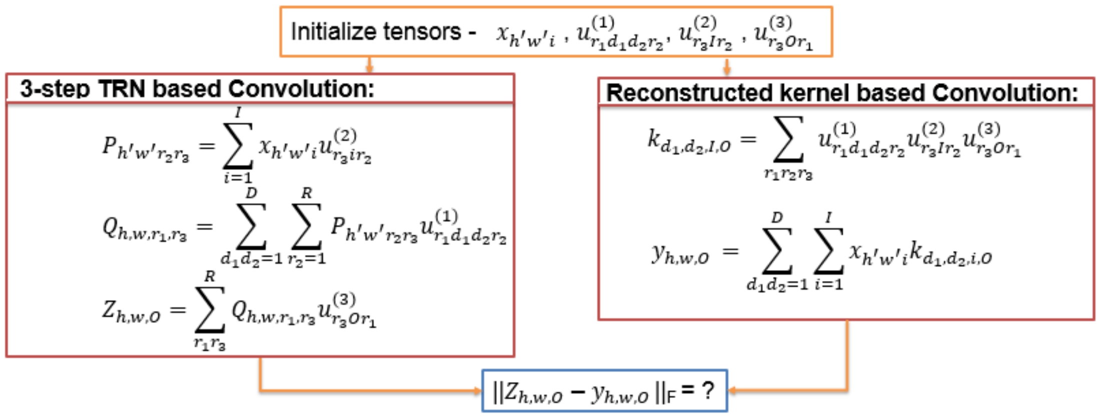
At the end of every unit, the image size is reduced by half and the number of layer filters is increased by 2x. The filter size is 3x3 in all the convolution layers. The filters increase from 16 in initialization layer to 640 in the final convolution layers. In total, all the filters of the network has 36.5 million trainable parameters.

4. RESULTS AND COMPARISONS

4.1 Layer-level Results: TRN based Convolution

4.1.1 Verification

The implementation of TRN based convolution scheme in python needs to be verified before integrating it in the WideResnet Code. For this, we compare the output tensors from following two methods, as shown in the Figure 4.1. Here, the reconstructed-kernel based method constructs a kernel of uncompressed dimensions from factored tensors and then convolution is performed with that reconstructed kernel. Input tensor $\mathcal{X}$ and factored kernel tensors $u^{(i)}$ were randomly initialized and fed to both the methods.



Initialize tensors - $x_{h'w'i}$, $u^{(1)}_{r_1d_1d_2r_2}$, $u^{(2)}_{r_3Ir_2}$, $u^{(3)}_{r_3Or_1}$
3-step TRN based Convolution:
$P_{h'w'r_2r_3} = \sum_{i=1}^{I} x_{h'w'i}u^{(2)}_{r_3ir_2}$
$Q_{h,w,r_1,r_3} = \sum_{d_1d_2=1}^{D}\sum_{r_2=1}^{R} P_{h'w'r_2r_3}u^{(1)}_{r_1d_1d_2r_2}$
$Z_{h,w,O} = \sum_{r_1r_3}^{R} Q_{h,w,r_1,r_3}u^{(3)}_{r_3Or_1}$
Reconstructed kernel based Convolution:
$k_{d_1,d_2,I,O} = \sum_{r_1r_2r_3} u^{(1)}_{r_1d_1d_2r_2}u^{(2)}_{r_3Ir_2}u^{(3)}_{r_3Or_1}$
$y_{h,w,O} = \sum_{d_1d_2=1}^{D}\sum_{i=1}^{I} x_{h'w'i}k_{d_1,d_2,i,O}$
$||Z_{h,w,O} - y_{h,w,O}||_F$ = ?

Figure 4.1. Verification method for TRN implementation

The outputs from both methods i.e $\mathcal{Z}_{h',w',O}$ and $\mathcal{Y}_{h,w,o}$ should be equivalent. The equivalency is checked via Frobenius norm of the difference of the two tensors given by $||\mathcal{Z}_{h',w',O} - \mathcal{Y}_{h,w,o}||_F$. Table 4.1 shows the equivalence for different input-filters and output-filters.

Table 4.1.
Frobenius Norm of the difference of layer outputs for different layer sizes

Layer input	**Uncompressed Kernel dimensions**	**Compressed(TRN) Kernel dimensions**	Frobenius Norm for Rank = 3	Frobenius Norm for Rank = 7
100x32x32x16	3x3x16x16	3x3x(4x2x2)x(4x2x2)	0.00591	0.02834
100x32x32x32	3x3x32x64	3x3x(4x4x2)x(4x4x4)	0.02099	0.03101
100x32x32x64	3x3x64x64	3x3x(4x4x4)x(4x4x4)	0.01051	0.04057
100x32x32x640	3x3x640x640	3x3x(10x8x8)x(10x8x8)	0.05268	0.05731

The Frobenius norm is near zero in all the cases and hence the TRN based Convolution is implemented correctly. In case of mismatch, a neat trick is to initialize the factored tensors with different ranks so that an error would be triggered if the tensor merging isn't appropriate .

4.1.2 Runtime comparison with regular convolution

Table 4.2 compares, for different layer sizes, the flops of a regular convolution with TRN based 3-step convolution. The filter size, denoted by 'D', is 3 while the stride is 1. The rank for TRN convolution is chosen such that its flops are almost equal to the uncompressed convolution flops. It follows that any lower rank should provide a computational speed-up.

Table 4.3 shows the wall-clock time in a controlled Tensorflow environment i.e., without intra_op and inter_op_parallelism. It can be observed that the speed-up is achieved for a slightly lesser rank than what the speed-up rank should be from flops computation. This is due to the fact that there exists unaccounted overhead like

Table 4.2.
Comparison of flops of a regular convolution with TRN based 3-step convolution

Layer Input (B x H x W x I)	**Kernel Size (D x D x I x O)**	**Regular Convolution Flops** $[BHWD^2IO]$	**TRN based Convolution flops** $[HW(4R^3(I+O)+BR^2(I+O)+BR^3D^2)]$
100x32x32x16	3x3x16x16	2.359296 x 10^8	2.13504 x 10^8 (for Rank = 5)
100x32x32x16	3x3x16x32	4.718592 x 10^8	4.18480 x 10^8 (for Rank = 6)
100x32x32x64	3x3x640x64	3.774873 x 10^9	2.75660 x 10^9 (for Rank = 10)
100x32x32x640	3x3x640x640	3.774873 x 10^{11}	5.02963 x 10^{10} (for Rank = 15)
100x32x32x640	3x3x640x640	3.774873 x 10^{11}	2.84405 x 10^{11} (for Rank = 30)

the slicing & reshape operations during the merging of factored tensors. In addition, when a layer is narrow (filters = 16 or 32), the compute doesn't match, indicating that the compression framework is more suitable for wide layers (filters = 64 and above).

4.1.3 Runtime comparison for a parallel implementation

In the preceding section, all the r_3 convolutions in each step of TRN based convolution (as described in Figure2.8) were executed sequentially. Hence, if the r_3 convolutions are implemented in parallel then it results in additional reduction in computational complexity.

Figure 4.2 shows the 'timeline' view of a layers convolution for both sequential and parallel implementations. The input and filter are two randomly initialized tensors of dimensions 100x32x32x64 and 3x3x64x64 respectively. The TRNs rank is 8 and so is the number of convolutions in each step. Hence, r_3 is 8. The runtime in both cases are as follows:

1. Sequential implementation: Run-time = 415 ms

Table 4.3.
Comparison of runtime of a regular convolution with TRN based 3-step convolution

Layer Input (B x H x W x I)	**Kernel Size (D x D x I x O)**	**Regular Convolution Runtime (ms)**	**Regular Convolution Runtime without intra_op and inter_op_parallelism (ms)**	**TRN based Convolution Runtime without intra_op and inter_op_parallelism (ms)**
100x32x32x16	3x3x16x16	0.0212 +/- 0.0005	0.0466 +/- 0.0023	0.0710 +/- 0.0009 (for Rank = 2)
100x32x32x16	3x3x16x32	0.0285 +/- 0.0004	0.0739 +/- 0.0037	0.0826 +/- 0.0010 (for Rank = 2)
100x32x32x64	3x3x64x64	0.1666 +/- 0.0011	0.4257 +/- 0.0054	0.4175 +/- 0.0075 (for Rank = 8)
100x32x32x640	3x3x640x640	12.508 +/- 0.4309	39.455 +/- 0.5316	7.5149 +/- 0.0686 (for Rank = 16)
100x32x32x640	3x3x640x640	12.508 +/- 0.4309	39.455 +/- 0.5316	19.943 +/- 0.2715 (for Rank = 28)

2. Parallel implementation: Run-time = 143 ms

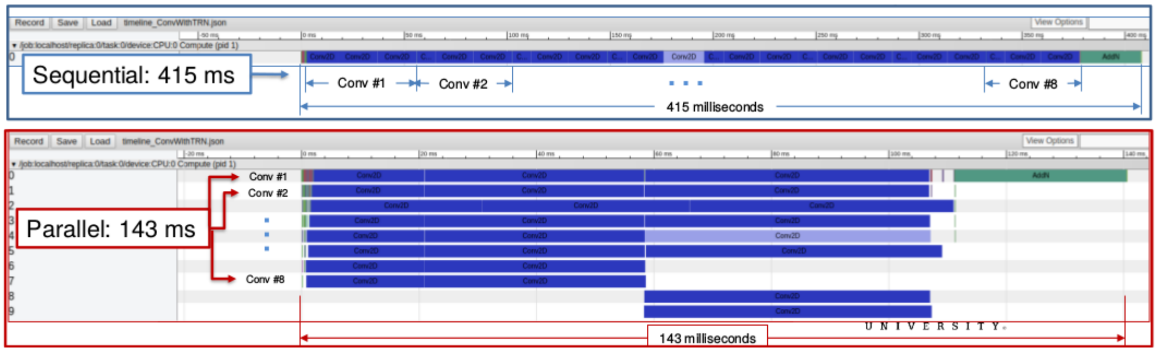


Figure 4.2. Sequential vs Parallel runtime of a TRN based convolution

Table 4.4 compares runtime of a parallel implementation with different layer sizes. It can be noticed that the runtime reduction is maximum when layer is wide(filters = 64 or 640) and the convolution time, shown by blue rectangles in figure 4.2, doesn't reduce linearly with the number of parallel threads. Further, the convolution operation itself takes more time than usual when steps are implemented in parallel.

4.2 Network-level Results: Compressed WideResnet Results

We utilize the 3-step TRN based convolution function to train WideResnet on CPU in a restricted Tensorflow environment wherein the functions that control the parallel implementation of Tensorflow graph nodes are exposed to the end-user. This restriction is essential to test our hypothesis that the TRN based convolution is computationally efficient than the regular convolution.

The network parameters were randomly initialized from a Gaussian distribution. Specifically, as suggested by Wang et al., (2018); the tensor factors were initialized with a variance $\sigma = (\frac{2}{N})^{\frac{1}{d}} \frac{1}{\sqrt{R}}$ [13], where N is the number of parameters in the uncompressed network, d is the number of factored tensors and R is the rank of the tensors.

Table 4.4.
Comparison of runtime between a sequential and parallel implementations of TRN based 3-step convolution

Layer Input (B x H x W x I)	**Kernel Size (D x D x I x O)**	**TRN based Convolution Runtime (ms) (Sequential r_3 convolutions)**	**TRN based Convolution Runtime (ms) (Parallel r_3 convolutions)**
100x32x32x16	3x3x16x16	0.0710 +/- 0.0009 (R = 2)	0.0411 +/- 0.0011 (R = 2)
100x32x32x16	3x3x16x32	0.0826 +/- 0.001 (R = 2)	0.0481 +/- 0.0016 (R = 2)
100x32x32x64	3x3x64x64	0.4175 +/- 0.0075 (R = 8)	0.1429 +/- 0.0016 (R = 8)
100x32x32x640	3x3x640x640	7.5149 +/- 0.0686 (R =16)	2.7041 +/- 0.0192 (R =16)

All the experiments on WideResnet were implemented on a desktop with Core i7 920 CPU and an Nvidia GTX 1070 GPU. Specifically, the network training was carried out on the GPU to obtain the accuracy results while the profiling results (both test and train times) were obtained on CPU. Further, all the results for the experiments in the subsequent sections were obtained with the following settings:

- All the networks were trained to 200 epochs with a minibatch size of 100.
- The models were trained using Stochastic Gradient Descent (SGD) with momentum 0.9 and a decaying learning rate.

Since TRN based convolution does not use kernel of regular shape but a compressed kernel, the table 4.5 shows the compressed kernel's composite tensor shape for various layers of WideResnet architecture [13].

Table 4.5.
Regular kernel and compressed TRN kernel shapes for WideResnet Architecture

Layer	**Uncompressed Kernel dimensions**	**Compressed(TRN) Kernel dimensions**
Initialization	3 x 3 x 3 x 16	9 x 3 x (4 x 2 x 2)
Unit1	ResBlock(3 x 3 x 16 x 160)	9 x (4 x 2 x 2) x (8 x 5 x 4)
	ResBlock(3 x 3 x 160 x 160) x 4	9 x (8 x 5 x 4) x (8 x 5 x 4)
Unit2	ResBlock(3 x 3 x 160 x 320)	9 x (8 x 5 x 4) x (8 x 8 x 5)
	ResBlock(3 x 3 x 320 x 320) x 4	9 x (8 x 8 x 5) x (8 x 8 x 5)
Unit3	ResBlock(3 x 3 x 320 x 640)	9 x (8 x 5 x 4) x (10 x 8 x 8)
	ResBlock(3 x 3 x 320 x 320) x 4	9 x (10 x 8 x 8) x (10 x 8 x 8)

4.2.1 Single rank for all layers

The layer-wise flops for ever other layer in the 28-layer WideResnet is shown in the table 4.6 and pictorially in the figure 4.3. In computing the flops the batch size was assumed to be 100 and the filter size to be 3x3.

Table 4.6.
Comparison of flops of an uncompressed WideResnet with a compressed WideResnet

Layer Number	**Layer Name and Size**	**Regular WideResnet flops**	**Compressed WideResnet flops (Rank=21)**
0	ResUnit_0_32x32x3x16	4.42E+07	1.011E+10
1	ResUnit_1_32x32x16x160	2.36E+09	2.316E+10
3	ResUnit_1_32x32x160x160	2.36E+10	3.512E+10
5	ResUnit_1_32x32x160x160	2.36E+10	3.512E+10
7	ResUnit_1_32x32x160x160	2.36E+10	3.512E+10
9	ResUnit_2_16x16x160x320	1.18E+10	1.210E+10
11	ResUnit_2_16x16x320x320	2.36E+10	1.543E+10
13	ResUnit_2_16x16x320x320	2.36E+10	1.543E+10
15	ResUnit_2_16x16x320x320	2.36E+10	1.543E+10
17	ResUnit_3_8x8x320x640	1.18E+10	5.519E+09
19	ResUnit_3_8x8x640x640	2.36E+10	7.181E+09
21	ResUnit_3_8x8x640x640	2.36E+10	7.181E+09
23	ResUnit_3_8x8x640x640	2.36E+10	7.181E+09
	Half Network's Sum	**2.38E+11**	**2.24E+11**

Figure 4.4 compares wall-clock time of every layer in a uncompressed regular WideResnet with a compressed WideResnet. Here, the rank of all compressed layers

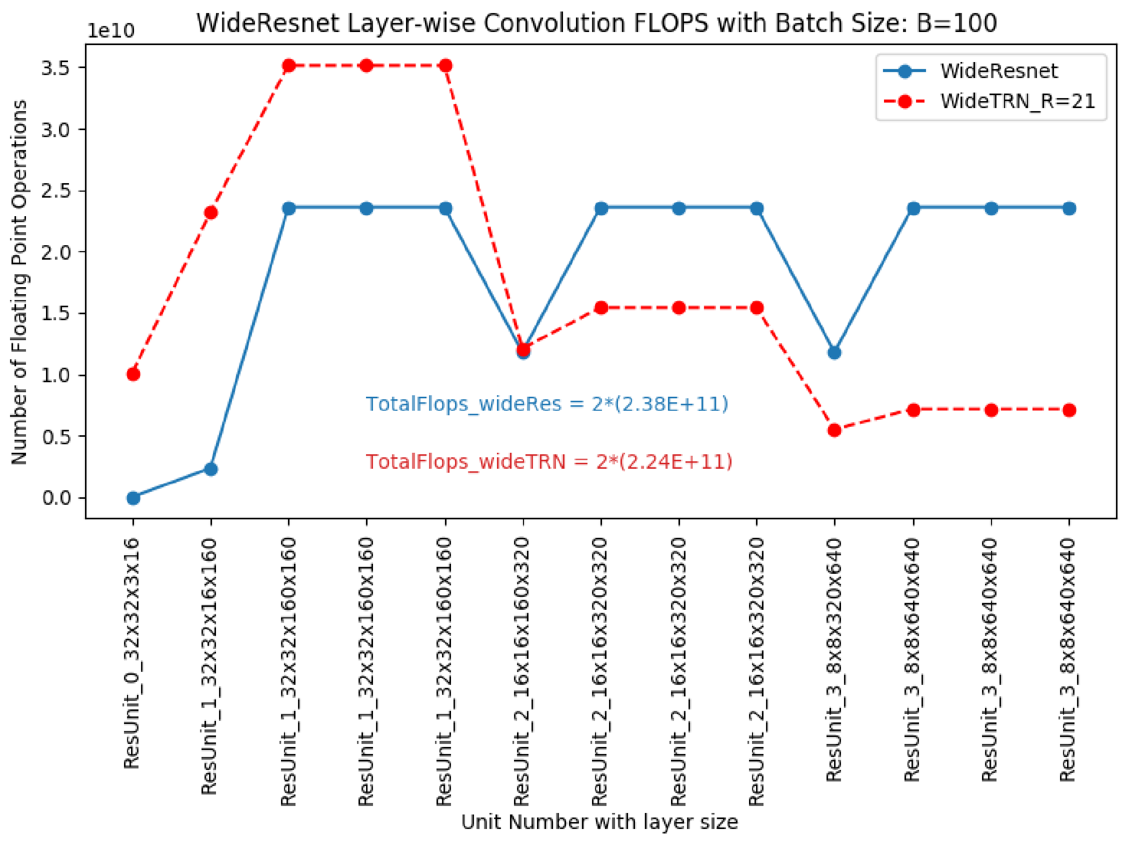


Figure 4.3. Comparison of flops for every other layer in an uncompressed WideResnet and a compressed WideResnet

was chosen to be 20. It can be observed that the test (or evaluation) time is lower for compressed WideResnet [≈ 56seconds] when compared to uncompressed WideResnet [≈ 72 s].

The higher runtime for all the layers in Unit 1 is due to the fact that all of its layers are narrow [# of filters = 160] in relative to the chosen rank.

Table 4.7 compares both training and testing time for different ranks. The batch size for training is 128 and for testing it is 100. It can be observed that when R=8, the training time reduces by 3x and the evaluation or test time reduces by 4x.

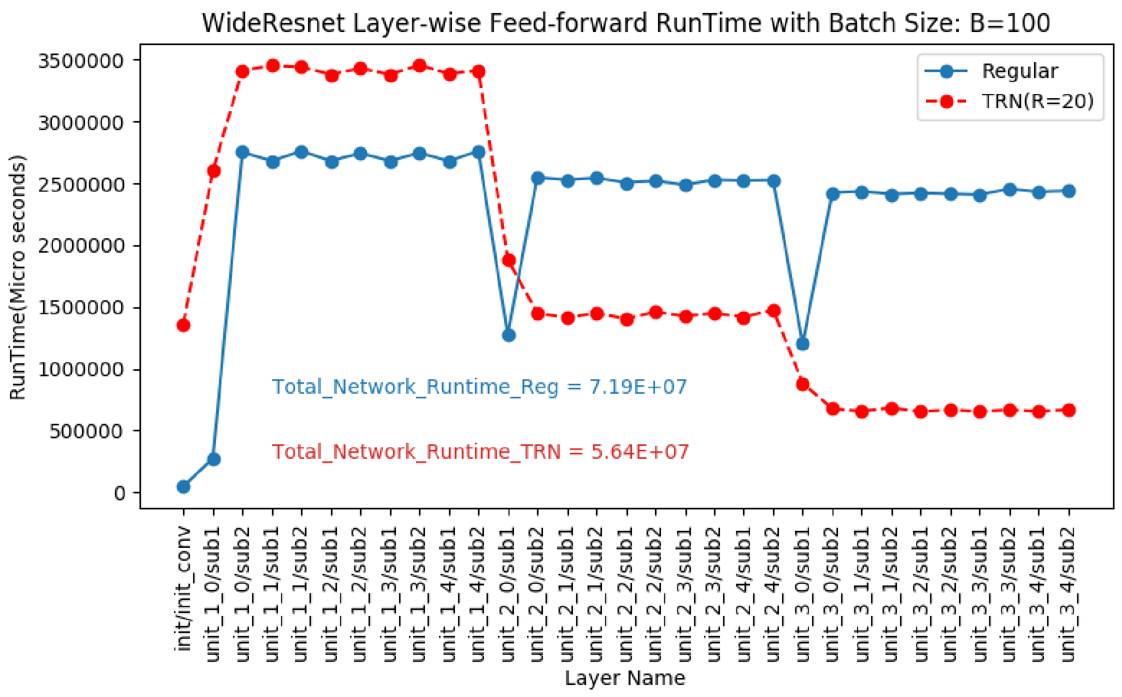


Figure 4.4. Comparison of an uncompressed WideResnet runtime with a compressed WideResnet

4.2.2 Selective rank for individual layers

It can be observed that for compressed networks with large ranks, the initial layers possess a predominantly high runtime. With an aim of reducing the compute time further, especially for higher ranks where the accuracy level doesn't degrade much, different ranks were chosen for the initial few layers. Figure 4.5 shows the layer-wise runtime for a sample scenario when the first four layers were chosen a different rank from the rest of the layers. Table 4.8 provides a overview of the network performance for such selective rank selection for individual layers.

It can be observed that when R=2, the network is faster to infer by almost 2x while the degradation in accuracy is minimal at around 1 percentage points.

Table 4.7.
Comparison of runtime of an uncompressed WideResnet with a compressed WideResnet for different ranks

Network type	**Average train time per iteration (s)**	**Average test time per iteration (s)**	**Accuracy (%)**
WideResnet	265.639 +/- 0.9655	60.433 +/- 0.2486	78.2
TRN (R = 20)	245.831 +/- 0.8786	52.6521 +/- 0.1562	74.1
TRN (R = 8)	79.426 +/- 0.1882	14.9604 +/- 0.0645	70.4
TRN (R = 4)	51.6711 +/- 0.2259	8.7673 +/- 0.0429	66.2
TRN (R = 2)	41.2004 +/- 0.0807	6.6674 +/- 0.04150	57.5

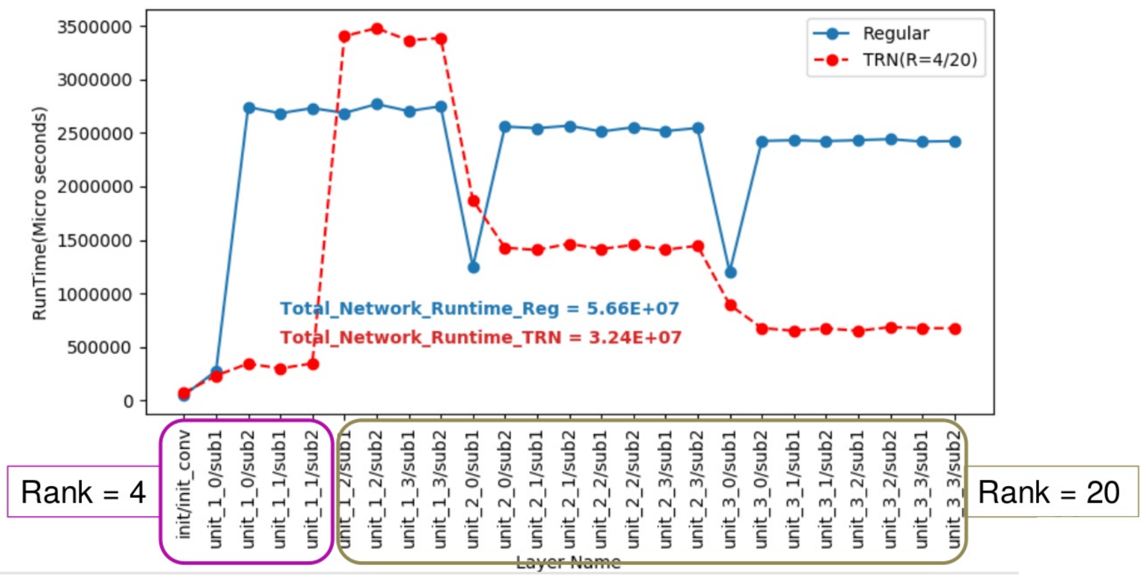


Figure 4.5. Comparison of runtime of an uncompressed WideResnet with a targeted rank based compressed WideResnet

4.2.3 Low ranks for a 2x deep network

As the computational intensity is reduced With a TRN based compressed network it is informative to investigate the improvement in accuracy if the network depth is

Table 4.8.
Compressed WideResnet performance for layer-wise rank selection

Rank for the first 4 layers	**Rank for the rest of the layers**	**Average train time per iteration (s)**	**Average test time per iteration (s)**	**Accuracy**
R = 2	R = 20	182.716 +/- 1.2218	38.2914 +/- 0.1779	73.0 %
R = 4	R = 20	185.116 +/- 1.5726	38.9732 +/- 0.1452	73.2 %
R = 8	R = 20	193.493 +/- 1.3249	40.7007 +/- 0.0860	73.5 %
R = 20	R = 20	245.831 +/- 0.8786	52.6521 +/- 0.1562	74.1 %

increased. As a case study, we consider a compressed network with twice the depth of a WideResnet. The number of residual units in the network is doubled while retaining the same number of input/output filters per unit. A comparison of the network layer configuration is shown in the table 4.9. The trade-off between computational time and accuracy for such a network is shown in the table 4.10.

Table 4.9.
Network layout of a 2x deep WideResnet

Layer type	**Regular WideResnet Dims (28-layer)**	**Deep WideResnet Dims (52-layer)**
Init	3x3x16x16	3x3x16x16
Unit1	ResBlock(3, 16, 160)	ResBlock(3, 16, 160)
	[ResBlock(3, 160, 160)]x3	[ResBlock(3, 160, 160)]x7
Unit2	ResBlock(3, 160, 320)	ResBlock(3, 160, 320)
	[ResBlock(3, 320, 320)]x3	[ResBlock(3, 320, 320)]x7
Unit3	ResBlock(3, 320, 640)	ResBlock(3, 320, 640)
	[ResBlock(3, 640, 640)]x3	[ResBlock(3, 640, 640)]x7

Table 4.10.
52-layer Compressed WideResnet performance on CIFAR-100 dataset

Rank for all the layers	**Average train time per iteration (s)**	**Average test time per iteration (s)**	**Accuracy**
R = 2	80.923 +/- 1.1354	12.7743 +/- 0.0279	61.1 %
R = 4	100.448 +/- 1.3427	16.9765 +/- 0.1127	66.9 %
R = 8	153.714 +/- 1.2836	29.6324 +/- 0.1577	71.2 %

5. SUMMARY

The current work proposes a targeted rank-selection framework that compresses a wide convolution neural network optimally and explores parallelizability of a novel within-layer parallel processing design to train neural networks. This framework utilizes the Tensor Ring Network(TRN) based decomposition for significant network compression whilst simultaneously reducing the computational intensity (both in inference and training), with minimal loss in accuracy. The proposed framework can be implemented in various platforms with configurable parallel computing environment such as smart-phones and CPU clusters. To demonstrate the performance and energy efficiency, we applied the framework to optimally compress WideResnet architecture. Our results indicate that the framework achieves high computation efficiency with minimal degradation in performance.

6. FUTURE WORK

The current framework procured the runtime information while training the network on CPU, since a restricted Tensorflow environment is essential and the functions that control the parallel implementation of Tensorflow graph nodes are exposed to the end-user on CPU (not on GPU). Hence, as a subsequent step this framework can be implemented on a GPU.

The restrictions on Tensorflow environment may be relaxed to determine the practical gain in computation-time when huge number of parallel computing resources are at the researcher's disposal. As a first step, regular convolution flops have to be updated to account for Intra_Op_Parallelism. In the subsequent step, the researcher has to pool adequate parallel cores and then introduce both intra_Op_Parallelism and inter_Op_Parallelism in a measured manner to obtain maximum speed-up.

REFERENCES

REFERENCES

[1] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[4] Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 1578–1585. IEEE, 2017.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] Misha Denil, Babak Shakibi, Laurent Dinh, Nando De Freitas, et al. Predicting parameters in deep learning. In *Advances in neural information processing systems*, pages 2148–2156, 2013.

[8] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[9] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

[10] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[11] Christopher M Bishop et al. Pattern recognition and machine learning (information science and statistics). 2006.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[13] Wenqi Wang, Yifan Sun, Brian Eriksson, Wenlin Wang, and Vaneet Aggarwal. Wide compression: Tensor ring nets. *learning*, 14(15):13–31, 2018.

[14] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[15] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[16] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[17] Jean Kossaifi, Zachary C Lipton, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. Tensor regression networks. *arXiv preprint arXiv:1707.08308*, 2017.

[18] Wenqi Wang, Vaneet Aggarwal, and Shuchin Aeron. Efficient low rank tensor ring completion. *Rn*, 1(r1):1, 2017.

[19] Google LLC. Tensorflow, an open source machine learning framework, 2015.

[20] Stack Exchange Inc. Stack overflow, hottest 'tensorflow' answers, 2015.

[21] Alex Krizhevsky. Cifar-10 and cifar-100 datasets, 2012.

[22] Resnet Research models. Tensorflow models on resnet, 2017.

[23] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.