Tamp: A Library for Compact Deep Neural Networks with Structured Matrices

Bingchen Gong*
Zhejiang University
gbc@zju.edu.cn

Brendan Jou, Felix Yu, Shih-Fu Chang
Columbia University
{bjou,yuxinnan,sfchang}@ee.columbia.edu

ABSTRACT

We introduce **Tamp**, an open source C++ library for reducing the space and time costs of deep neural network models. In particular, **Tamp** implements several recent works which use structured matrices to replace unstructured matrices which are often bottlenecks in neural networks. **Tamp** is also designed to serve as a unified development platform with several supported optimization back-ends and abstracted data types. This paper introduces the design and API and also demonstrates the effectiveness with experiments on public datasets.

1. INTRODUCTION

Deep neural networks have achieved outstanding performances in many fields over the past several years. However, training and inference of most neural networks often have high computational costs preventing many popular neural network architectures, particularly those for visual recognition, from being deployed on-board in resource-limited devices such as mobile phones, low cost robots and other embedded devices. One major bottleneck comes from the fully-connected layers which compute the matrix vector product followed by an element-wise activation function: $\phi(\mathbf{Cx})$. For a layer with a k-dimension input and n-dimension output, the linear transformation matrix has nk parameters and usually costs nk operations to compute.

One family of promising methods for reducing the memory and time costs of fully connected layers is to impose structure on the matrices used in these networks [1, 7, 10, 11]. By replacing unstructured dense matrices with *structured matrices* the number of network parameters can be dramatically reduced, with very modest accuracy loss. For example, by imposing the [1] structure on the AlexNet architecture [5], one can get 4,000× space saving and 1.38× speedup with a less than a 2% cost to accuracy on ILSVRC2012 [9]. The intuition behind the "compressibility" of these networks is that there exist redundancies in the dense matrices. However, many of these proposed approaches are based on closed-source implementations or are reported on different software architectures. This makes real-world application and benchmarking across such methods difficult.

In this open source submission, we introduce **Tamp**[1], an open-sourced C++ library providing a unified platform to develop, benchmark and deploy structured matrices in deep neural networks. The library supports using two popular frameworks, Caffe [3] and TensorFlow [2], as back-ends.

2. STRUCTURED TRANSFORMS

We briefly summarize several different structures in **Tamp**. Note that the projection matrix is restricted to be square by some of these methods, requiring pre or post-processing for input or output vectors.

Circulant [1, 11]**.** Given an n-dimensional real-valued vector $\mathbf{v}$ as the first column vector of it, a corresponding circulant matrix $\mathbf{C} \in \mathbb{R}^{n\times n}$ has the structure:

$$\mathbf{C} = \mathrm{circ}(\mathbf{v}) = \begin{bmatrix} v_0 & v_{n-1} & \dots & v_2 & v_1 \\ v_1 & v_0 & v_{n-1} & & v_2 \\ \vdots & v_1 & v_0 & \ddots & \vdots \\ v_{n-2} & & \ddots & \ddots & v_{n-1} \\ v_{n-1} & v_{n-2} & \dots & v_1 & v_0 \end{bmatrix}. \quad (1)$$

With a circulant matrix as the matrix of parameters in a fully-connected layer, for an input data vector $\mathbf{x}$, the layer output $\phi(\mathbf{Cx})$ can be calculated efficiently using the Fast Fourier Transform (FFT):

$$\phi(\mathbf{Cx}) = \phi(\mathbf{ifft}(\mathbf{fft}(\mathbf{v}) \circ \mathbf{fft}(\mathbf{x}))), \quad (2)$$

where **fft** and **ifft** denote FFT and inverse FFT, respectively; $\circ$ corresponds to Hadmard (elementwise) product between two vectors. This FFT equivalence allows us to compute circulant projections in $\mathcal{O}(n \log n)$ time $\mathcal{O}(n)$ space. In practice, a random sign flipping diagonal matrix $\mathbf{D}$ is added between $\mathbf{C}$ and $\mathbf{x}$ to improve the capacity of the model [1].

Toeplitz-like [11]**.** A Toeplitz-like matrix is composed by f-circulant matrix, a generalization of circulant matrices defined by a real-valued scalar f:

$$\mathbf{Z}_f(\mathbf{v})_{i,j} = \begin{cases} \mathrm{circ}(\mathbf{v})_{i,j} * f & : i < j \\ \mathrm{circ}(\mathbf{v})_{i,j} & : i \geq j \end{cases} \quad (3)$$

*This work was done in part while the first author was an undergraduate visiting researcher at Columbia University's Digital Video & Multimedia Lab.

MM '16, October 15 - 19, 2016, Amsterdam, Netherlands

ISBN 978-1-4503-3603-1/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2964284.2973802

[1] https://github.com/ColumbiaDVMM/Tamp

Method	Complexity	Method	Complexity
Unstructured	$\mathcal{O}(nk)$	Low Rank [10]	$\mathcal{O}(r(n+k))$
Circulant [1]	$\mathcal{O}(n \log n)$	Toeplitz [11]	$\mathcal{O}(n \log n)$
Skew-circulant [11]	$\mathcal{O}(n \log n)$	ACDC [7]	$\mathcal{O}(n \log n)$

Table 1: Summary of Computational Complexities of Several Structured Matrix Transforms.

When f is 1, the f-circulant matrix is the same as a circulant matrix and when f is -1, it is a skew-circulant matrix. Order-r Toeplitz-like matrices then are all $n \times n$ matrices which can be composed by two f-circulant matrices [11] as:

$$\mathbf{C}_r(\mathbf{G}, \mathbf{H}) = \sum_{i=1}^{r} \mathbf{Z}_1(\mathbf{g}_i)\mathbf{Z}_{-1}(\mathbf{h}_i), \tag{4}$$

where $\mathbf{G} = [g_1 \dots g_r]$, $\mathbf{H} = [h_1 \dots h_r] \in \mathbb{R}^{n \times r}$. For most space complexity saving, order-1 Toeplitz-like matrices are used in this initial release. Similar to circulant matrices, FFTs can be used to speed up computation.

Low Rank [10]**.** For a matrix in the form of $\mathbf{C} \in \mathbb{R}^{n \times k}$ whose rank is r, there exist two matrices $\mathbf{A} \in \mathbb{R}^{n \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times k}$ such that $\mathbf{C} = \mathbf{AB}$. By this factorization, the number of parameters reduces from nk to $r(n+k)$, and the time complexity reduces from nk to $r(n+k)$. It is also shown recently that tensor factorization, a generalization of low-rank factorization, can also be used in compressing neural networks [8].

ACDC [7]**.** With $\mathbf{A} = [a_1 \dots a_r]$, $\mathbf{D} = [d_1 \dots d_r] \in \mathbb{C}^{n \times r}$ to scale the signal in the original domain and the Fourier domain respectively [7], an order-r AFDF transformation replaces linear projection in neural networks as

$$\mathbf{AFDF}_r(\mathbf{x}) = \mathbf{x}^T[\prod_{i=1}^{r} \text{diag}(\mathbf{a}_i)\mathcal{F}\text{diag}(\mathbf{d}_i)\mathcal{F}^{-1}]$$

where $\mathcal{F}$ denotes the FFT matrix, so $\mathcal{F}\mathbf{x}$ and $\mathcal{F}^{-1}\mathbf{x}$ correspond to $\mathbf{fft}(\mathbf{x})$ and $\mathbf{ifft}(\mathbf{x})$. AFDF can be viewed as a further generalization of circulant structures. ACDC is the real-valued version of AFDF, where the Fourier transform is replaced by the Discrete Cosine Transformation (DCT) [7].

3. LIBRARY FEATURES

3.1 Interfaces & Tools

Tamp is a C++ library for implementing, benchmarking and easy execution of structured transforms in deep neural networks. It provides convenient interfaces via C++ template classes, abstracts device/host memory management, and BLAS and FFT library.

To create a new structured matrix model in **Tamp**, develop a subclass of the `ProcessorBase` class and override forward/backward calculation functions like below.

The `Forward` function computes the feedforward stage of the structured matrix model, and the `Backward` function computes gradients of the model. These two functions can be overloaded to add support for different acceleration architectures, e.g. GPUs or coprocessors.

An `Environment` struct is passed to the constructor containing model parameters such as the output dimensions and whether to include a random sign flipping diagonal matrix $\mathbf{D}$. To summarize, the following steps are necessary to develop a new structured matrix model with **Tamp**:

```
class Processor : ProcessorBase
{
  Processor(Environment* env) :
    ProcessorBase(env) {}
  void Forward(
    ExecutiveCore* core,
    const ProcessorTape* atape)
  { /* Forward calculation */ }
  void Backward(
    ExecutiveCore* core,
    const ProcessorTape* atape,
    const ProcessorTape* btape)
  { /* Gradient calculations */ }
  void Shape(
     ExecutiveCore* core,
     ProcessorTape* atape)
  { /* Set output size */ }
};
```

- Override `Forward` and `Backward` calculation methods.
- Override the `Shape` method. This method is called each time before forward calculation to set the output size and allocate the proper buffer size.

The following additional steps are optional for developers to provide additional acceleration support like on GPUs:

- Implement a constructor to set the inner class status.
- Overload `Forward` and `Backward` calculation methods to support acceleration.

3.2 Data Type Abstraction

Although not required, inheriting from `ProcessorBase` by a template class is preferred since **Tamp** is designed for different data types. Using templates, a `Processor` can be registered as:

```
template <typename T>
class Processor : ProcessorBase
{
  /* Functions to override */
}
INSTALL_PROCESSOR(float, Processor<float>)
INSTALL_PROCESSOR(double, Processor<double>)
```

Most of the classes and math functions in **Tamp** are implemented as templates. One of them, `TypedData<T>`, class template abstracts data management. `TypedData<T>` is a pure virtual class that inherits from `BufferedData` and uses backend-specific implementations such as `TypedDataTensor<T>` and `TypeDataCaffe<T>`.

3.3 Context Management

Because of the multi-threaded nature of TensorFlow [2], member functions of `ProcessorBase` may be called concurrently, necessitating context management. **Tamp** provides a class `ExecutiveCore` as an interface for current execution context and uses the `ProcessorTape` class to record data. Although Caffe [3] will not call instances concurrently, similar context management is provided for coherence.

Through the `ProcessorTape` and `ExecutiveCore` classes, developers can allocate memory buffers and limit their access within context to avoid the use of mutexes.

```
void Forward(
  ExecutiveCore* core,
  const ProcessorTape* atape)
{
  const TypedData<float>& input =
      atape->input[0]->typed<float>();
  TypedData<float>& output =
      atape->output[0]->typed<float>();
  ...
  auto conv_buffer =
    core->allocateBuffer<float>({3,3});
  ...
}
```

3.4 Math Library

By using standard C pointers for data I/O interface, libraries are kept lightweight and easy to use in **Tamp**. Most functions use a CBLAS back-end and a small subset of FFT operations require libfftw3. For efficient CPU operations, we use various libraries like ATLAS/FFTW/MKL/etc, while for GPU, we use NVIDIA's cuBLAS and cuFFT along with CUDA. Also noteworthy, in Caffe [3], their portable header `math_functions` conveniently makes all math function templates type independent.

3.5 Backend Differences

Tamp currently integrates both TensorFlow [2] and Caffe [3], and so any structured matrix models implemented in **Tamp** can be run with a TensorFlow or Caffe back-end. Although we have abstracted most of the back-end interfacing to developers, some nuanced differences exist when using each of these back-ends due to their inherent differences.

TensorFlow [2]**.** A model is defined via two "operators" in TensorFlow, a forward and backward operator. The backward operator is bound to a forward operator through a Python wrapper using a `RegisterGradients` decorator [2]. Technically, developers can override the forward method, then implement backward code in Python and register them manually; however, doing so prevents the model from being compatible with Caffe [3].

```
# Python definition of network: Circulant section
weights = _variable_
biases = _variable_
circulant = struct.circulant(pool, weights) + biases
```

Calling the structured model like a normal TensorFlow operation from this point on will then work normally.

Caffe [3]**.** In Caffe, a structured matrix model corresponds to a single layer, and can be defined for example as:

```
layer {
  name: "circulant"
  type: "CirculantProjection"
  bottom: "pool"
  top: "circulant"
}
```

In Caffe, biases are implemented within layers, thus they can not be added outside the model. **Tamp** provides a trivial biases which can be added to model automatically.

Along with the **Tamp** library, we release two basic structure implementation: Circulant and Skew-circulant. These two models can be used directly or can be combined to form more complex structures, like Toeplitz-like matrices [11].

4. BENCHMARKS & PERFORMANCE

We used **Tamp** to implement multiple structured matrices and benchmarked them using both Caffe [3] and TensorFlow [2] on MNIST, CIFAR10 and ILSVRC2010. The experiments were run on a Intel(R) Xeon(R) CPU E5-2609 @ 2.40GHz.

MNIST. The MNIST handwritten digits dataset contains 60K/10K 28×28 greyscale train/test examples. We trained the convolutional neural network LeNet [6] here, replacing the second-to-last fully-connected layer with structured matrices. Results are given in Table 2.

		Caffe [3]		TensorFlow [2]	
Method	**#Params**	**Acc**	**Speed**	**Acc**	**Speed**
Unstructured	250,000	99.07	798	99.20	131
Circulant [1]	500	98.84	797	99.00	138
Skew-circulant [11]	500	98.75	703	98.90	130

Table 2: *Tamp* Accuracy ("Acc" in %) and Speed (examples/sec) Benchmarks on MNIST.

CIFAR10. In the CIFAR10 dataset, there are 50K/10K 32×32 color train/test examples from ten classes [4]. We trained a CIFAR10-CNN network on CIFAR10 [4], where we replaced the only fully-connected layer with structured transforms. Our CIFAR10 results are shown in Table 3. Note that the structured matrices should often be used to structure intermediate fully-connected layers not the last fully-connected layer providing classification output. In the case of CIFAR10-CNN, however, we found that introducing structure in the last layer also leads to minimal compromise on accuracy.

		Caffe [3]		TensorFlow [2]	
Method	**#Params**	**Acc**	**Speed**	**Acc**	**Speed**
Unstructured	10,240	82.53	1111	86.00	621
Circulant [1]	1,024	80.35	1176	83.70	680
Skew-circulant [11]	1,024	79.77	1136	81.20	671
Low Rank [10]	6,144	71.82	1153	78.70	690
Toeplitz [11]	2,048	81.03	1142	81.40	643
ACDC [7]	2,048	81.08	1153	81.50	658

Table 3: *Tamp* Accuracy (%) and Speed (examples/sec) on CIFAR10. Number of parameters is also shown for one fully-connected layer.

ILSVRC. We trained AlexNet [5] on ILSVRC2010 [9]. Our results show a ∼90% reduction in the memory requirement of AlexNet using a circulant structure in exchange for ∼2% increase in top-1 and top-5 error rate. Here, we replaced the two fully-connected layers before the last fully-connected layer containing 4096 output units with circulant transforms. Results are shown below in Table 4.

	Top-5 Err	**Top-1 Err**	**Memory**
Unstructured	17.1%	42.8%	233.2 MB
Circulant [1]	19.4%	44.1%	20.5 MB

Table 4: *Tamp* Benchmarks of Circulant Structures [1] in AlexNet [5] on ILSVRC10 [9].

5. CONCLUSIONS

We introduces a C++ library **Tamp** as a unified platform for constructing structured matrices to replace dense linear transformation operations in neural networks. Developers can easily utilize either Caffe [3] or TensorFlow [2] deep network back-end to develop, benchmark and deploy compressed networks. As part of the initial release, we provide implementations of several structured matrices including circulant, low rank, Toeplitz-like matrices and the ACDC transform. **Tamp** uses templates to implement data abstraction and type independent math operations, and also provides context management to support concurrent computing. In particular, our experiments on MNIST, CIFAR10 and ILSVRC verify the claims of structured matrices in neural networks for reducing the number of parameters with modest cost to accuracy compared to unstructured transforms. We believe that our work will enable repeatable research and enable neural network applications with limited resource.

6. REFERENCES

[1] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *ICCV*, 2015.

[2] Google, Inc. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. http://tensorflow.org.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM Multimedia*, OSSC, 2014.

[4] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11), 1998.

[7] M. Moczulski, M. Denil, J. Appleyard, and N. de Freitas. ACDC: A structured efficient linear layer. In *ICLR*, 2016.

[8] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov. Tensorizing neural networks. In *NIPS*, 2015.

[9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *arXiv:1409.0575*, 2014.

[10] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *ICASSP*, 2013.

[11] V. Sindhwani, T. N. Sainath, and S. Kumar. Structured transforms for small-footprint deep learning. In *NIPS*, 2015.