



Deep³: Leveraging Three Levels of Parallelism for Efficient Deep Learning

Bitu Darvish Rouhani¹, Azalia Mirhoseini², Farinaz Koushanfar¹

¹University of California San Diego, ²Rice University
bitu@ucsd.edu, azalia@rice.edu, farinaz@ucsd.edu

ABSTRACT

This paper proposes Deep³, an automated platform-aware Deep Learning (DL) framework that brings orders of magnitude performance improvement to DL training and execution. Deep³ is the first to *simultaneously* leverage three levels of parallelism for performing DL: *data*, *network*, and *hardware*. It uses platform profiling to abstract physical characterizations of the target platform. The core of Deep³ is a new extensible methodology that enables incorporation of platform characteristics into the higher-level data and neural network transformation. We provide accompanying libraries to ensure automated customization and adaptation to different datasets and platforms. Proof-of-concept evaluations demonstrate 10-100 fold physical performance improvement compared to the state-of-the-art DL frameworks, e.g., TensorFlow.

1. INTRODUCTION

Deep Learning (DL) has become the key methodology for achieving the state-of-the-art inference performance by moving beyond traditional linear or polynomial analytical machine learning [1]. Despite DL's powerful learning capability, the computational overhead associated with DL methods has hindered their applicability to resource constrained settings. Examples of such settings range from embedded devices to data centers where physical viability (e.g., runtime and energy efficiency) is a standing challenge.

To optimize DL physical performance, there are at least two sets of challenges that should be addressed simultaneously. The first challenge set has to do with the costly iterative nature of DL training process [1]. What signifies the relevance of this cost is the empirical context of DL algorithms: even though DL neural networks have shown superior results compared to their linear machine learning counterparts, their success has been mainly based on experimental evaluations with the theoretical aspects yet to be developed. As such, reducing the DL performance cost, particularly training runtime, enables realization of different DL models within the confine of the available computational resources to empirically identify the best one.

The second challenge set is related to the mapping of computations to increasingly multi-core and/or heterogeneous modern architectures. The cost of computing and moving data to/from the memory and inter-cores is different for each computing platform. The existing solutions for performing DL are divided into two distinct categories: (i) Data scientists, on the one hand, have been mainly focused on opti-

mizing DL efficiency through data and neural network manipulations and pruning with little or no attention to the platform characterization [2, 3, 4]. (ii) Computer engineers, on the other hand, have developed hardware-accelerated solutions for an efficient domain-customized realization of DL models, e.g., [5, 6]. Although these works demonstrate significant improvement in the realization of particular DL models, to the best of our knowledge, none of the prior works have looked into the end-to-end solution on how the higher level data-dependent signal transformation can benefit the physical performance. Our hypothesis is that devising platform aware signal transformations could highly favor the underlying learning task by holistic customization to the limits of the hardware resources, data, and DL neural network.

We propose Deep³, the first automated end-to-end DL framework that explicitly optimizes the physical performance cost by higher-level DL transformation. Deep³ provides performance metrics to quantify DL overhead in terms of (i) the number of arithmetic operations dedicated to training/executing DL networks, and (ii) the number of (inter-core and inter-memory) communicated words. Common performance indicators such as runtime, energy, and memory footprint can be directly deduced from these metrics. To optimize for these costs, we introduce a systematic approach for automated platform customization as well as projection error tuning while achieving the target accuracy.

Deep³ optimization is devised based on *three inter-linked* thrusts. First, it identifies the intrinsic platform characteristics by running a set of subroutines to find the optimal local neural network size that fits the resource constraints (*Hardware Parallelism*). Second, it proposes a novel DL graph traversal methodology that leverages the fine- and coarse-grained parallelism in the data, neural network, and physical platform for efficient hardware implementation. Deep³ transforms the global DL training task into the parallel training of multiple smaller size local DL networks while minimally affecting the accuracy. Our approach balances the trade-off between memory communication and local calculations to improve the performance of costly iterative DL computations (*Network Parallelism*). Third, it leverages a new data and resource aware signal projection as a pre-processing step to reduce the input data feature space size customized to the local neural network topology dictated by the platform (*Data Parallelism*). Deep³ exploits the degree of freedom in producing several possible projection embeddings to automatically adapt the data and DL circuitry to the physical limitations imposed by the platform.

The explicit contributions of this paper are as follows:

- Proposing Deep³, the first end-to-end DL framework which is simultaneously data, neural network, and hardware parallel. Deep³ achieves significant performance cost reduction by platform aware signal transformation while delivering the same inference accuracy compared to the state-of-the-art DL approaches.
- Incepting a novel neural network graph traversal methodology for efficient mapping of computations to the limits of the target platform. Our approach is scal-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '17, June 18-22, 2017, Austin, TX, USA

© 2017 ACM. ISBN 978-1-4503-4927-7/17/06...\$15.00

DOI: <http://dx.doi.org/10.1145/3061639.3062225>

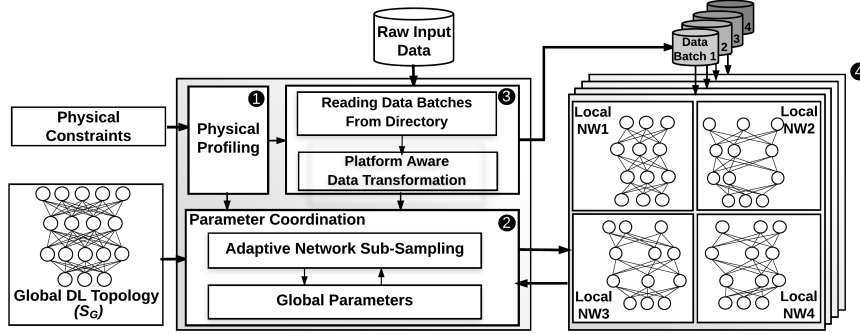


Figure 1: High level block diagram of Deep³ framework. Deep³ leverages the data, neural network, and hardware fine- and coarse-grained parallelism to adaptively customize DL in accordance to the physical resources and constraints.

able while it greatly reduces costly memory interactions in performing DL training and execution.

- Developing performance cost metrics and automated platform aware signal transformations to optimize the data projection and DL neural network architecture for the underlying resources and physical constraints.
- Devising accompanying libraries to ensure Deep³ ease of adoption on three common classes of platforms including CPUs, GPUs, and FPGAs. Note that our proposed methodology is universal and directly applicable to other families of computing hardware.
- Creating customized approaches to provide support for *streaming scenarios* where data dynamically evolves over time. Proof-of-concept evaluations for both static and dynamic data corroborate 10-100 fold performance improvement compared to the prior art solutions.

2. PRELIMINARIES

Deep learning is a hierarchical learning mechanism that is designed to extract meaningful abstractions from raw data with limited required human interactions [1]. Table 1 summarizes common layers used in DL neural networks. To train a DL model, each batch of data is processed in two main steps: (i) forward propagation, and (ii) backward propagation. In the forward propagation, the raw values of data features are fed into the first layer of the network (input layer). These raw features are gradually mapped to higher-level abstractions based on the current values of the DL parameters as outlined in the Table 1. The acquired data abstractions are then used to predict the inference label in the last layer of the DL network (output layer).

In the backward propagation, a batch *gradient-based* algorithm is applied to *fine-tune* the DL network parameters so that a specified loss function is minimized. The loss function captures the difference between the neural network inference (output of forward propagation) and the ground-truth labeled data. In particular, DL parameters are updated per:

$$W_{ij}^{(s)} = W_{ij}^{(s)} - \eta \frac{1}{b_s} \sum_{k=1}^{b_s} \frac{\partial E^{(s)}}{\partial W_{ij}^{(s)}}, \quad (1)$$

where η is the learning rate, b_s is the data batch size, and $(\partial E^{(s)} / \partial W_{ij}^{(s)})$ represents the propagated errors in the layer s . The forward and backward propagations are *iteratively* applied for multiple rounds of *reprocessing* the input data until the desired accuracy is achieved. Once the DL neural network is fully trained, the execution phase only includes one round of forward propagation for each data sample. In the rest of the paper, we mainly focus our discussions on

training DL models as it is known to be more computationally expensive. However, we emphasize that our methodology ultimately benefits the DL execution as well by reducing the data and DL circuitry customized to the platform.

3. Deep³ GLOBAL FLOW

Figure 1 illustrates the high-level block diagram of Deep³ for training DL networks. Deep³ takes the stream of data samples and the *global DL topology* (S_G) as its input. S_G is a user-defined vector of integers whose components indicate the number of neurons per hidden layer of the global DL model. Deep³ consists of four main modules to learn the global parameters of a DL network customized to the limits of the physical resources and constraints:

(i) **Physical profiling:** Deep³ characterizes the target platform by running subroutines that contain basic operations involved in the forward and backward propagations (Section 4). The characterization is a one-time process and incurs a fixed negligible overhead.

(ii) **Parameter coordination:** Deep³ uses the output of physical profiling as guidelines to map the global DL training into smaller size local neural networks that fit the platform constraints. The local networks are formed by subsampling the neurons of the global DL model based on an *extensible depth-first* graph traversal methodology that we introduce to minimize the communication overhead per training iteration (Section 5). Each local neural network is then replicated into multiple computing kernels that are executed in parallel using different data batches.

(iii) **Platform aware data transformation:** Deep³ leverages a new resource aware signal projection as a pre-processing step. It transforms the stream of input data into multiple lower-dimensional subspaces that fit the local network topology dictated by the platform. Deep³ uses the degree of freedom in producing several possible projection embeddings in order to customize costly DL training/execution to the limits of the platform and data structure (Section 6).

(iv) **Local network updating kernels:** Each local neural network updates a fraction of the global model using different data batches. Each fraction has the same depth as the global network with far fewer edges. The local updates are periodically aggregated through the parameter coordination unit to optimize the weights of the global model (Section 5).

We provide accompanying libraries to ensure Deep³ ease of use for data scientists and engineers. Our libraries provide support for training/execution of DL models on multi- and many-core CPU, CPU-GPU, and CPU-FPGA platforms.

4. HARDWARE PARALLELISM

The structure of a local network (i.e., the number of neurons per layer) has a direct impact on the corresponding memory footprint and overall system performance. Deep³

Table 1: Commonly used layers in DL networks.

DL Layer	Description	Computation
Convolution	Multiplying the filter weights (W_{ij}) with the post-nonlinearity values in the preceding layer (y_{ij}^{l-1}) and summing the results	$x_{ij}^{(l)} = \sum_{a=1}^m \sum_{b=1}^m W_{ab}^{(l-1)} \times y_{(i+a)(j+b)}^{l-1}$
Max Pooling	Computing the maximum value of $k \times k$ overlapping regions in the $N \times N$ grid of the underneath layer	$x_{ij}^{(l)} = \text{Max}(y_{(i+a)(j+b)}^{l-1})_{a \in \{1, 2, \dots, k\}, b \in \{1, 2, \dots, k\}}$
Mean Pooling	Computing the mean value of $k \times k$ non-overlapping regions in the $N \times N$ grid of the underneath layer	$x_{ij}^{(l)} = \text{Mean}(y_{(i+a)(j+b)}^{l-1})_{a \in \{1, 2, \dots, k\}, b \in \{1, 2, \dots, k\}}$
Fully-Connected	Multiplying the weights of the l^{th} layer (W_{ij}^l) with the post-nonlinearity values in the preceding layer (y_i^{l-1})	$x_i^{(l)} = \sum_{j=1}^{n_{l-1}} W_{ij}^{(l-1)} \times y_j^{(l-1)}$
Non-linearity	Sigmoid	$S_j(x_j) = \frac{1}{1+e^{-x_j}}$
	Softmax	$\sigma_j(x_j) = \frac{e^{x_j}}{\sum_{k=0}^{n_c} e^{x_k}}$
	Tangent Hyperbolic (Tanh)	$\frac{\sinh(x)}{\cosh(x)}$
	Rectified Linear Unit (ReLU)	$\text{Max}(0, x)$

provides a set of subroutines that characterize the impact of neural network size on the subsequent resource consumption. Our subroutines measure the performance of basic operations involved in the DL training/execution (Table 1). Examples of such operations include convolution, matrix multiplication, non-linearities, and inter-core communication.

Note that the realization of DL operations can be highly diverse depending on the target platform. For instance, based on the dimensionality of the matrices being multiplied, a matrix multiplication can be compute-bound, bandwidth-bound, or occupancy-bound on a specific platform. Our subroutines run such operations with varying sizes to spot the target platform constraints.

Deep³ takes a user-defined performance metric such as runtime, energy, energy \times delay, and/or memory as its input. To optimize for the target performance metric, Deep³ uses the output of physical profiling as guidelines to break down the global DL model into subsequent local neural networks that fit the pertinent resource provisioning (Section 5). The physical profiling unit outputs a vector of integers, S_L , that has the same length as the input global model S_G . Each element of S_L indicates the maximum size of the corresponding layer that fits into the platform. Platform characterization is a one-time process and incurs a fixed negligible overhead compared to the DL training task. We use vendor supplied libraries to model the target platform as suggested in [7].

5. NEURAL NETWORK PARALLELISM

Traditionally, a *breadth-first* (a.k.a., stripe-based) model partitioning is used to distribute DL training workload, e.g., TensorFlow. Figure 2a illustrates the conventional approach used for neural network parallelism. In this setting, the activations of neural connections that cross the machine boundaries should be exchanged between different nodes to complete one round of forward and backward propagation. What exacerbates the DL training cost in this context is the variance in the processing time of different nodes. To mitigate the effect of this variation, authors in [2, 4, 8] suggest the use of asynchronous partial gradient updates for DL training. These works achieve meaningful runtime reduction by parallelism in the training of DL networks with sparse connectivity. However, for fully-connected graphs fine-tuning the parameters is still a bottleneck given the greedy layer-wise nature of DL and the higher communication overhead of fully-connected neural networks.

Deep³, for the first time, proposes a *depth-first graph traversal* methodology to distribute DL training workload customized to the platform constraints. As shown in Figure 2b, Deep³ transforms the global DL model into multiple overlapping local neural networks that are formed by subsampling the neurons of the global network. Each local neural network has the same depth as the global model with far fewer edges. Our approach minimizes the communication overhead per training iteration. This is because each local

network can be updated independently without having to wait for partial gradient updates from successive layers to be communicated between different computing nodes. The local updates are periodically aggregated through the parameter coordinator to optimize the weights of the global model. The number of neurons per local network, S_L , is a design configuration dictated by the physical resources (i.e., memory bandwidth, cache size, or available energy).

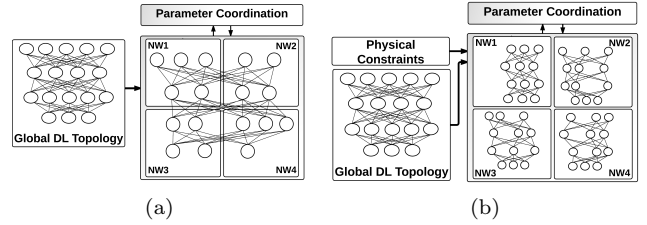


Figure 2: Network parallelism: The existing breadth-first DL partitioning approaches require communicating the partial updates multiple times for processing each batch of data. On the contrary, our depth-first approach limits such communications to only once after multiple training iterations.

5.1 Parameter Coordination

Algorithm 1 outlines the pseudocode of Deep³ parameter coordination unit and local DL computing kernels. Let us denote the parameter coordination unit with $P_{id} = 0$. For each available computing kernel (e.g., an FPGA, GPU, or CPU core), the parameter coordinator initiates a pair of *send- and receive-thread*. The send-thread subsamples the neurons of the global DL model in accordance with the local DL topology, S_L , dictated by the platform (Section 4). It communicates the selected subsample of the global model to its associated computing kernel for further processing. Each send-thread keeps track of the selected subsample of DL neurons indices ($index_L$) sent to its associated computing kernel in a list called *history*. Storing the indices is required to later aggregate the updated parameters in the global model.

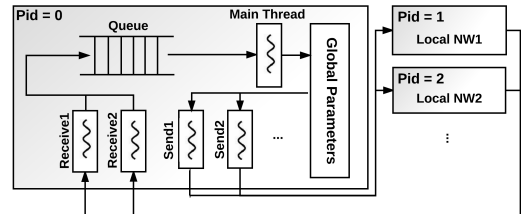


Figure 3: Flow of data in Algorithm 1.

The receive-thread reads back the computed gradients from each local computing kernel. It enqueues the local gradients (ΔW_L) along with the *threadID* and the number of communications occurred between the coordinator and that specific kernel so far. This information is then used by the

main thread in the coordination unit to retrieve the corresponding global indices from the history list and aggregate the partial local gradients into the global DL model. The send and receive threads keep working until they got interrupted by the main thread once the specified error threshold is met or a certain number of DL iterations have been performed. Figure 3 illustrates the flow of data in Algorithm 1.

Algorithm 1 Deep³ Neural Network Parallelism

Input: Global DL Model (S_G), Error Threshold δ , Local DL Model (S_L), Training Data Embedding & Labels (C_{Tr}), Number of Processors N_p , and Maximum Number of Iterations Max_itr

Output: Trained Global DL Parameters DL_{glob}

```

0. Send_Thread (threadID,  $S_G$ ,  $S_L$ , history) :
    send_count = 0
    While(!done_flag) :
        IndexL ← NetworkSubsampling( $S_L$ )
         $DL_{local}^{init}$  ←  $S_G.get\_weights(Index_L)$ 
        comm.send( $DL_{local}^{init}$ , dest = threadID)
        history[threadID].append(IndexL)
        send_count ← send_count + 1

1. Recieve_Thread (threadID,  $Q$ ,  $Q\_Lock$ ) :
    Rcounter = 0
    While(!done_flag) :
         $\Delta W_L$  = comm.recv(source = threadID)
        lock( $Q\_Lock$ )
         $Q.put([\Delta W_L, threadID, Rcounter])$ 
        release( $Q\_Lock$ )
        Rcounter ← Rcounter + 1

2. Main :
    if ( $P_{id} == 0$ ) : //Parameter Coordinator
         $Q = Queue()$ 
        queue_Lock = threading.Lock()
         $DL_{glob} \leftarrow RandomInitialization()$ 
        itr = 0
        history = []
        done_flag ← False
        Creation & Initialization of Send_Threads
        & Receive_Threads
    // Main Thread in Parameter Coordinator
    While( $\tilde{\delta} \geq \delta$  or itr ≤ Max_itr) :
        [ $\Delta W_L, threadID, Rcounter$ ] ←  $Q.get()$ 
        IndexL ← history[threadID][Rcounter]
         $DL_{glob} \leftarrow S_G.get\_weights(S_G)$ 
         $S_G.set\_weights(DL_{glob} + \Delta W_L, Index_L)$ 
         $\tilde{\delta} \leftarrow UpdateValidationError(S_G)$ 
        itr ← itr + 1
        done_flag ← True
         $DL_{glob} \leftarrow S_G.set\_weights(S_G)$ 
        Broadcasts done_flag & Exit
    else: //Local Network Kernels
        While(!done_flag) :
             $DL_{local}^{init} = comm.recv(source = 0)$ 
             $S_L.set\_weights(DL_{local}^{init})$ 
             $DL_{local} \leftarrow DL_{local}^{init}$ 
            i ← 0
            While (i ≤  $n_{push}$ ) :
                 $C_{Tr}^i \leftarrow GetNextDataBatch()$ 
                 $DL_{local} \leftarrow UpdateDL(DL_{local}, C_{Tr}^i)$ 
                i ← i + 1
             $\Delta W_L \leftarrow DL_{local} - DL_{local}^{init}$ 
            comm.send( $\Delta W_L$ , dest = 0)

```

In Deep³, each local network is independently trained using different data batches generated by the data transformation unit (Section 6). A local network might compute its gradients (updates) based on a set of parameters that are slightly out of date. This is because the other local networks

have probably updated the global values in the parameter coordination unit in the meantime. The impact of using stale parameters eliminates over time. The reason behind this is that the process of updating DL weights is *associative* and *commutative* per Eq. 1, and after several iterations, the DL parameters converge to the desired accuracy.

5.2 Computation-Communication Trade-off

Deep³ characterizes the time per training iteration for updating the global neural network as:

$$T_{itr} = n_{push} \underbrace{\sum_{i=1}^{N_p} (T_i^{FP} + T_i^{BP})}_{\text{Computation Cost}} + \underbrace{\frac{2}{n_{push}} \sum_{i=1}^{N_p} T_i^{comm}}_{\text{Communication Cost}}, \quad (2)$$

where the first term represents the computation cost and the latter term characterizes the inter-core communication overhead. We use T_i^{FP} and T_i^{BP} to denote the forward and backward propagation costs and N_p to represent the number of concurrent processors. The variable n_{push} indicates the frequency of model aggregation (i.e., the number of training data batches that should be processed before a local network pushes back its updates to the coordination unit). Table 2 details the computation and communication cost of training fully-connected Deep Neural Networks (DNNs). Similar setup applies to the Convolutional Neural Networks (CNNs) in which a set of convolutions are performed per layer. Deep³ finds an estimation of the physical coefficients listed in Table 2 by running a set of subroutines as discussed in Section 4. Our libraries provide support for both DNN and CNN models.

Table 2: Local Computation and Communication Costs.

Computation and Communication Costs
$T_i^{FP} = \alpha_{flop} \sum_{s=1}^{S-1} n_i^{(s)} n_i^{(s+1)} + \alpha_{act} \sum_{s=1}^S n_i^{(s)}$ S : total number of DNN layers α_{flop} : multiLadd + memory communication cost α_{act} : activation function cost
$T_i^{BP} = 2\alpha_{flop} \sum_{s=1}^{S-1} n_i^{(s)} n_i^{(s+1)} + \alpha_{err} \sum_{s=1}^S n_i^{(s)}$ α_{err} : propagation error cost
$T_i^{Comm} = \alpha_{net} + \frac{N_{bits} \sum_{s=1}^{S-1} n_i^{(s)} n_i^{(s+1)}}{BW_i}$ α_{net} : constant network latency N_{bits} : number of signal representation bits BW_i : operational communication bandwidth

The communication overhead in Deep³ is dominated by the cost of reading/writing the weights of local neural networks back and forth between the parameter coordinator and the computing kernels. On the one hand, a high value of n_{push} reduces the communication cost, but it also increases the computational load as the local networks are not frequently combined to optimize the global DL model. On the other hand, a low value of n_{push} degrades the training performance due to a significant increase in the communication overhead. Deep³ tunes the variable n_{push} accordingly to fit the physical limitations while balancing computation versus communication.

6. DATA PARALLELISM

The input layer size of a neural network is conventionally dictated by the feature space size of the input data samples used for DL training. Deep³'s data transformation module takes the local DL topology (S_L) imposed by the platform as its input and aims to find the lower-dimensional data embedding that best matches the dictated local model. It works by factorizing the raw input data $A_{m \times n}$ into a *dictionary matrix* $D_{m \times l}$ and a *data embedding* $C_{l \times n}$ such that:

$$\underset{l, D_{m \times l}, C_{l \times n}}{\text{minimize}} (\delta_{valid}^{local}) \quad s.t. \quad \|A - DC\|_F \leq \epsilon \|A\|_F, \quad l \leq m, \quad (3)$$

where δ_{valid}^{local} is the partial validation error acquired by training the local neural networks using the projected data embedding C instead of the raw data A . $\|\cdot\|_F$ denotes the Frobenius norm and ϵ is an intermediate approximation error that casts the rank of the input data.

Eq. 3 is a part of an overall objective that we aim to optimize in order to train/execute a DL model within the given computational resources. To solve Eq. 3, we first initiate the matrices D and C as empty sets. Deep³ gradually updates the corresponding data embeddings by streaming the input data as outlined in Figure 4. In particular, for a batch of newly arriving samples (A_i), Deep³ first calculates a projection error, $V(A_i)$, based on the current values of the dictionary matrix D . This error shows how well the newly added samples can be represented in the space spanned by D . If the projection error is less than a user-defined threshold (β), it means the current dictionary D is good enough to represent those new samples (A_i). Otherwise, Deep³ modifies the corresponding data embeddings to include the new data structure imposed by the recently added samples. Our data projection approach is linear in complexity and incurs a negligible overhead as we experimentally verify in Section 7.

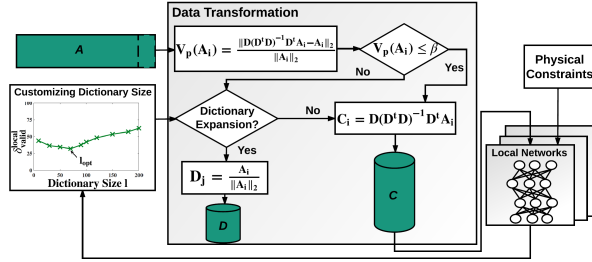


Figure 4: Data Parallelism: Deep³ maps the stream of input data to a corresponding lower-dimensional embedding in linear time. The data embedding is used to update the local DL models.

Note that after adding enough samples to the dictionary D , little improvement is observed in the DL accuracy as a result of increasing the size of the DL input layer. While checking the projection error $V(A_i)$ ensures that Deep³ doesn't add linearly-correlated samples to the dictionary, getting feedback from the DL model prevents unnecessary increase in the size of the neural network's input layer.

Many complex modern datasets that are not inherently low-rank can be modeled by a composition of multiple lower-rank subspaces [9, 10, 11]. Methods such as Principal Component Analysis (PCA) are oblivious to the coarse-grained parallelism existing in the data; they always return a unified subspace of data equal to the rank of the matrix. Deep³ leverages a new composition of high-dimensional dense data as an ensemble of multiple lower-dimensional subspaces by carefully selecting dictionary samples from the data itself rather than using principal vectors. This type of composition has been suggested in the literature to facilitate knowledge extraction [9] or improve the system's physical performance [10, 11]. However, no earlier work has adapted the alignment of data lower-dimensional subspaces as a way to facilitate global training of large-scale DL networks customized to the local DL circuitry that best matches the target platform. In addition, traditional projection methods such as PCA incur a quadratic complexity, which makes it a costly choice for projecting large datasets.

7. EXPERIMENTS

We provide accompanying libraries for our realization of Deep³ framework. Our implementations for multi-core CPU

and CPU-GPU platforms are built to work with highly popular DL libraries, e.g., TensorFlow [8] and Theano [12]. In our FPGA realization, we use 16 bits fixed-point for DL computations. Each variable is represented in two's complement format using 1 sign bit, 3 integer bits, and 12 fraction bits. We choose the fixed-point format for our FPGA implementation for two main reasons: (i) The range of parameters within a DL model is bounded due to applying non-linear activation functions such as Tanh or Sigmoid. Thereby, the use of fixed-point format does not significantly degrade the computational accuracy as shown in [13]; (ii) Fixed-point implementation incurs a smaller area overhead and is significantly faster compared to its floating-point counterpart. We use floating-point format for our multi-core CPU and GPU realizations.

One common approach in training a DL model is the use of Stochastic Gradient Descent (SGD) to fine-tune the neural network parameters. In our realization of SGD, instead of using a fixed learning rate η in computing the local gradients, we used Adagrad technique [14] to adapt the learning rate of each parameter according to the update history of that parameter. In particular, Deep³ computes the learning rate of the i^{th} parameter at iteration K as $\eta_{iK} = \frac{\eta}{\sum_{j=1}^K \Delta W_{ij}^2}$, where ΔW_{ij} is the gradient of parameter i at iteration j .

7.1 Deep³ Performance Evaluation

Platform Settings. We evaluate Deep³ on three platforms: (i) **Platform 1** is a CPU-GPU co-processor with 192 CUDA cores and 4-Plus-1 quad-core ARM Cortex A15 CPU [15]. (ii) **Platform 2** is a CPU-FPGA setting in which a Xilinx Virtex-6 FPGA is hosted by an Intel core i7 processor. We use a 1Gbps Ethernet port to send data back and forth between the FPGA and the host. (iii) **Platform 3** is a multi-core CPU with an Intel core i7-2600K processor.

We based our evaluations on perceiving knowledge from (i) visual, (ii) smart-sensing, and (iii) audio data. Table 3 shows Deep³ total pre-processing time overhead. The pre-processing overhead can be broken down into the overhead of tuning the algorithmic parameters and data projection. The tuning itself accounts for both platform profiling and setting dictionary size l in accordance to the local neural network size dictated by the platform. We use a small subset of data (e.g., less than 5%) for tuning purposes. After fine-tuning the parameter l , the projection is performed using the customized value of l . In our implementation, all the CPU cores within each specified platform are employed for data projection using Message Passing Interface (MPI).

Table 3: Deep³ pre-processing overhead.

Application	Visual ^[16] (200 × 54129)			Smart-Sensing ^[17] (5625 × 9120)			Audio ^[18] (617 × 7797)		
	1	2	3	1	2	3	1	2	3
Platform ID	1	2	3	1	2	3	1	2	3
Tuning Overhead	49.7s	32.4s	63.5s	91.4s	53.8s	102.6s	31.1s	18.3s	37.5s
Data Projection Overhead	17.1s	9.8s	9.7s	18.9s	10.1s	10.1s	8.3s	4.9s	4.7s
Overall	66.8s	42.2s	73.2s	110.3s	63.9s	112.7s	39.4s	23.2s	42.2s

As our comparison *baseline*, we implement the state-of-the-art DL methodology in which dropout technique is used to boost the accuracy [19] and the global DL parameters are updated synchronously. We use Tanh as our activation function for each hidden layer, $\beta = 10\%$ as projection threshold, and $b_s = 100$ for SGD. For each of the visual, audio, and smart-sensing benchmarks, Table 4 reports the number of training iterations, time per iteration improvement, and the circuit footprint reduction achieved by Deep³ compared to the baseline approach on each of the platforms.

Note that although Deep³'s stochastic approach results in a higher *number of iterations* to train the global DL network within a specified accuracy compared to the baseline,

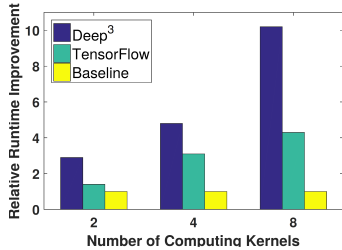
Table 4: Performance improvement achieved by Deep³ over prior-art deep learning approach.

Application	Visual (200 × 1000 × 300 × 9)			Smart-Sensing (5625 × 3000 × 500 × 19)			Audio (617 × 200 × 26)		
	CPU/GPU (Platform 1)	CPU/FPGA (Platform 2)	CPU-only (Platform 3)	CPU/GPU (Platform 1)	CPU/FPGA (Platform 2)	CPU-only (Platform 3)	CPU/GPU (Platform 1)	CPU/FPGA (Platform 2)	CPU-only (Platform 3)
DL Circuitry Reduction	4.3×	5.6×	5.4×	40.4×	191.5×	43.3×	6.2×	5.1×	6.2×
Number of Training Iterations	1.1×	1.7×	1.3×	1.8×	3.2×	2.1×	1.1×	1.4×	1.1×
Time Per Iteration Reduction	3.9×	5.7×	3.6×	11.4×	32.7×	9.2×	3.1×	4.1×	2.8×
Training Time Improvement	3.5×	3.3×	2.7×	6.3×	10.2×	4.3×	2.8×	2.9×	2.5×
Execution Time Improvement	1.2×	1.1×	1.2×	10.8×	9.7×	10.3×	5.9×	4.7×	5.6×

it gains significant overall runtime improvement by lessening the required *time per training iteration*. This improvement is achieved by customizing the data and DL circuitry to fit into the fast cache memory, avoiding the costly communication to the main memory of the platform. The training runtime improvement, in turn, also translates to significant savings in the energy consumption. We emphasize that the use of domain-customized DL accelerators such as Tensor Processing Unit (TPU) provide an orthogonal means for performance improvement. As such, Deep³ can achieve even further improvement by leveraging such accelerators.

In Table 4, the higher DL circuitry reduction for the CPU-FPGA setting is due to its limited available block RAM budget, which dictates a smaller local network compared to the other two platform settings. As we experimentally verify, although this DL circuitry reduction results in a higher number of iterations to converge to the same accuracy, Deep³ gains significant overall runtime improvement by reducing the required time per training iteration as a result of avoiding the communication with the off-chip memories.

Deep³ comparison with other DL frameworks. Figure 5 compares the runtime performance of Deep³ with the existing parallel DL solutions on a cluster of Intel core-i7 processors. In this experiment, we train Alexnet [20] with 60 million parameters using scaled CIFAR100 dataset. Deep³ shows excellent scaling pattern as the number of training cores increases. This is mainly because of the asynchronous depth-first nature of Deep³ framework that enables independent updating of local networks while balancing communication versus computation. Note that TensorFlow leverages an asynchronous breadth-first approach for distribution of DL workload (Section 5), and the baseline is devised based on a synchronous DL updating model as in Theano.

Figure 5: Deep³ relative runtime improvement.

8. CONCLUSION

We present Deep³, an automated end-to-end framework that provides holistic data and platform aware solutions and tools to efficiently perform DL training and execution. Deep³ leverages data, network, and platform customization to optimize DL physical performance. It maps the global DL training into smaller size local neural networks based on a novel extensible depth-first graph traversal methodology. Our approach minimizes the inter-core and inter-memory communication overhead while minimally affecting the accuracy. Our accompanying libraries automate adaptation of Deep³ for rapid prototyping of an arbitrary DL task. Our extensive evaluations show that Deep³ achieves significant improvements in DL training and execution.

Acknowledgments. This work was supported in parts by the ONR (N00014-11-1-0885) and NSF TrustHub (1649423) grants.

9. REFERENCES

- [1] L. Deng and D. Yu, “Deep learning: methods and applications,” *Foundations and Trends in Signal Processing*, 2014.
- [2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, “Large scale distributed deep networks,” *NIPS*, 2012.
- [3] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, “Deep learning with cots hpc systems,” *ICML*, 2013.
- [4] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, “Project adam: Building an efficient and scalable deep learning training system,” *USENIX OSDI*, 2014.
- [5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” in *FPGA*, 2015.
- [6] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, and E. Culurciello, “An efficient implementation of deep convolutional neural networks on a mobile coprocessor,” in *MWSCAS*, 2014.
- [7] Baidu DeepBench, <https://github.com/baidu-research/DeepBench>, 2016.
- [8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous systems, 2015,” *Software available from tensorflow.org*, vol. 1, 2015.
- [9] E. L. Dyer, A. C. Sankaranarayanan, and R. G. Baraniuk, “Greedy feature selection for subspace clustering,” *JMLR*, 2013.
- [10] A. Mirhoseini, B. D. Rouhani, E. M. Songhori, and F. Koushanfar, “Perform-ml: Performance optimized machine learning by platform and content aware customization,” *DAC*, 2016.
- [11] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, “DeLight: Adding energy dimension to deep neural networks,” *ISLPED*, 2016.
- [12] J. Bergstra, O. Breuleux, G. Bastien, J. Turian, D. Warde-Farley, and Y. Bengio, “Theano: a cpu and gpu math expression compiler,” *SciPy*, 2010.
- [13] A. W. Savich, M. Moussa, and S. Areibi, “The impact of arithmetic representation on implementing mlp-bp on fpgas: A study,” *IEEE Transactions on Neural Networks*, 2007.
- [14] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *JMLR*, 2011.
- [15] Jetson TK1, <https://developer.nvidia.com/jetson-tk1>, 2015.
- [16] Remote sensing, http://www.ehu.es/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes, 2015.
- [17] UCI Machine Learning Repository, <https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones>, 2015.
- [18] UCI Machine Learning Repository, <https://archive.ics.uci.edu/ml/datasets/isolet>, 2015.
- [19] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *JMLR*, 2014.
- [20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *NIPS*, 2012.