TinyDL: Just-In-Time Deep Learning Solution For Constrained Embedded Systems

Bita Darvish Rouhani[1], Azalia Mirhoseini[2], Farinaz Koushanfar[1]
[1]UC San Diego, [2]Rice University
bita@uscd.edu, azalia@rice.edu, farinaz@ucsd.edu

***Abstract*—This work proposes TinyDL, an automated end-to-end framework that aims to integrate the state-of-the-art Deep Learning (DL) models into embedded systems. TinyDL enables efficient training and execution of DL models as data is collected over time while adhering to the underlying physical resources and constraints. The constraints can be characterized in terms of memory bandwidth, energy resources (e.g., battery life), and/or real-time requirement. TinyDL takes advantage of platform profiling to abstract the physical characteristics of the target embedded device. We introduce a platform-aware signal transformation methodology to enable DL training and execution within the confine of the available resources. Our approach balances the trade-off between data movements and computations to improve the performance of costly iterative DL training/execution. Proof-of-concept implementation on NVIDIA Jetson TK1 embedded platform demonstrates up to two orders of magnitude energy improvement over the previous DL solutions, none of which had been amenable to constrained devices.**

I. Introduction

Sensor data has become an indisputable part of modern digital era that is changing the way people live and interact. The driving force behind several innovative sensing applications is the use of machine learning algorithms to infer behaviors and contexts from sensor data collected on portable devices. Inferring complex user behaviors from sensor measurements under real-world conditions, however, still remains brittle and unreliable. This, in turn, is acting as a bottleneck to sensing application's development, and makes it necessary to go beyond traditional linear or polynomial learning approaches to effectively model user behaviors.

Deep Learning (DL) is an emerging field of machine learning that has poised itself as the state-of-the-art approach in delivering robust and highly accurate inference in different learning domains [1]. Inspired by neural activities in the brain, deep learning models data through several successive layers of complex and non-linear features. As such, DL has the potential to overcome the challenges associated with noisy measurements, uncontrolled device positions, and intra-class diversity (e.g., the difference in data generated by diverse user populations) in sensing applications. While the nonlinear and sophisticated nature of DL empowers it to achieve extraordinary inference accuracy, it also brings new challenges concerning its scalability and resource utilization.

Given the rich set of embedded sensors in today's robots, mobile and wearable devices (e.g., accelerometers, gyroscopes, microphones, and cameras), and the great promise of deep learning, the ability to locally learn and infer sensing data undoubtedly provides a paradigm shift across a variety of domains, including health care, social networks, environmental monitoring, and transportation. There are several advantages in devising resource-efficient DL techniques that can be independently implemented on portable platforms: (i) Accessing the remote cloud servers is not always a possibility and even if so, it can be undesirably costly, specially since the DL model has to be continuously updated as new on-chip data is collected. (ii) Offloading data to the cloud brings major privacy concerns, particularly as many sensing data can reveal personal and private user information. (iii) Reducing the overhead of DL training and execution enables investigating several configurations of the acquired models which, in turn, leads to a more accurate inference.

This paper proposes TinyDL, a novel end-to-end framework which enables the first performance-efficient realization of deep learning on resource-constrained portable devices. The core of TinyDL framework is a new resource-aware signal transformation approach that adaptively maps the stream of input data to a corresponding ensemble of lower-dimensional subspaces. The dimensionality of input samples of a DL model has a direct impact on the overall size of the neural network, which subsequently dictates resource utilization for training and execution of DL models. The resource utilization can be characterized in terms of runtime, power, energy, and memory. Our signal transformation yields significant network compaction in accordance to the underlying resource provisioning while minimally affecting the inference accuracy.

TinyDL signal transformation is an adaptive pre-processing step that is amenable to large dynamic datasets. It works by factorizing the raw data matrix into a dictionary matrix D that includes a set of samples carefully selected from the input data, and a block-sparse coefficient matrix C where the *blocks* are organized such that the subsequent computations incur a minimal amount of memory access. The new representation highlights the most informative portions of the data, shrinking the DL training and execution workload. As a result, meaningful reductions in power and energy consumptions, memory footprint, and training/execution runtime are achieved. Our approach leverages the degree of freedom in producing several possible projection subspaces to enable customizing TinyDL with respect to the platform characteristics. We provide a systematic methodology to perform customization and projection error tuning to achieve a target accuracy. Note that our signal transformation is computed based on a streaming model which evades the requirement to store the original ever-growing data and incurs a fixed, low memory footprint.

II. TinyDL Framework

TinyDL leverages the hybrid structure of a dataset modeled as an ensemble of lower-dimensional subspaces to facilitate DL training and execution on resource-constrained platforms. TinyDL takes the stream of a large, dynamic dataset in

978-1-4673-6853-7/17/$31.00 ©2017 IEEE

the matrix form as its input and adaptively maps the data stream to a corresponding lower-dimensional embedding. As illustrated in Figure 1, TinyDL consists of three main units: (i) Automated customization, (ii) Signal transformation, and (iii) Deep learning. We have developed a user-friendly API that can be used to implement TinyDL on any CPU-only or any combined CPU-and-GPU SoC platforms for rapid prototyping of an arbitrary sensing application using DL models.

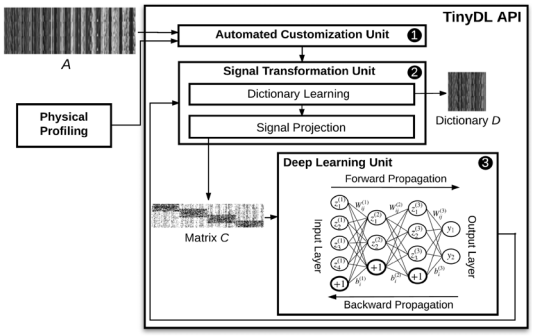

Fig. 1: High level block diagram of TinyDL.

A. Signal Transformation

signal transformation is a pre-processing step in TinyDL framework. Our goal is to tune TinyDL such that iterative deep network training using the transformed signal becomes much more efficient in terms of runtime, memory, and power compared to the conventional scenario where the raw data is fed into the DL model. Our key observation is that representing the data as an ensemble of lower-dimensional subspaces facilitates deep learning by (i) increasing the convergence rate, and (ii) reducing the overall DL network size in terms of the required number of neurons per layer.

To adaptively transform the stream of input data to a corresponding lower-dimensional embedding, TinyDL factorizes the data matrix A into a product of a dictionary matrix and a block-sparse coefficient matrix such that:

$$\underset{D\in R^{m\times l}, C\in R^{l\times n}}{\text{minimize}} \quad \|A-DC\|_F \quad \text{subject to} \quad \|C\|_0 \leq kn, \quad (1)$$

where $A_{m\times n}$ is the input data, $D_{m\times l}$ is the dictionary matrix, and $C_{l\times n}$ is the block-sparse coefficient matrix. $\|C\|_0$ measures the total number of non-zero elements in C, and k is the target sparsity level for each transformed signal ($k \leq l$). We show that $l \ll m \ll n$ can be achieved on real-world large datasets. The dictionary size l being less than the feature space size m is particularly of interest as it could be leveraged to adaptively transform the raw input data to a lower-dimensional embedding. TinyDL tailors the solution of Eq.1 for the underlying resource constraints to obtain the most accurate model in each application. Our approach incurs a low memory footprint and is well-suited for scenarios where storage is severely limited.

For each newly arriving sample (a_i), TinyDL first uses the current values of the dictionary matrix D to calculate a projection error denoted by $W_p(a_i)$. Next, it compares the calculated error with a user-defined projection threshold α and updates the corresponding lower-dimensional embedding if the projection error is less than or equal to α. In TinyDL, the dictionary matrix D is adaptively learned from the stream of input data and each column of the coefficient matrix C is computed using a greedy routine called Orthogonal Matching Pursuit (OMP). OMP is a well-known algorithm for solving sparse approximation problems. It takes a dictionary D and a sample a as inputs and iteratively approximates the sparse representation of the sample by adding the best fitting element in every iteration. Alg.1 demonstrates a pseudocode of the classic OMP algorithm. In OMP method, k denotes the total number of non-zeros per column of C (a.k.a., sparsity level). Once the block-sparse matrix C is computed, it is used as the input for training the selected DL network. Note that as we explain in Section II-C, TinyDL is devised with an automated resource-aware customization module that adaptively tunes the transformation parameters such as dictionary size l, and sparsity level k corresponding to the underlying constraints such as memory bandwidth and pre-processing runtime.

Algorithm 1 : OMP Algorithm

Inputs: Dictionary matrix D, Sample column a, Sparsity level k.
Output: Coefficient vector c.

1: $r^0 \leftarrow a$
2: $\Lambda^0 \leftarrow \emptyset$
3: **for *i = 1,...,k* do**
4: $\quad \Lambda^i \leftarrow \Lambda^{i-1} \cup argmax_j| < r^{i-1}, \ D_j > |$, *where j ranges over all the column index values of the dictionary D*
5: $\quad c_{\Lambda^i} \leftarrow argmin_c \|r^{i-1} - D_{\Lambda^i} c\|_2^2$, *where c is a vector variable in R^i*
6: $\quad r^i \leftarrow r^{i-1} - D_{\Lambda^i} c_{\Lambda^i}$
end for

B. Customized Deep Learning

After pre-processing the incoming samples, a DL-based classifier (e.g., Deep Neural Network (DNN)) is trained using the low-dimensional embedding of the data. The goal in DL training is to learn the weights and biases between layers such that a loss function is minimized. In our evaluations, we consider the L_2 norm difference between network inferences and the ground-truth labeled data as our loss function.

Training a DL model usually requires multiple paths through the whole dataset and comprises two main steps: (i) forward propagation, and (ii) backward propagation. In the forward propagation step, the model's response is computed based on the current values of the network parameters:

$$a_i^{(s+1)} = f(\sum_{j=1}^{n^{(s)}} W_{ij}^{(s)} a_j^{(s)} + b_i^{(s)}), \quad (2)$$

where $a_i^{(s)}$ is the state of unit i in layer s and $f(.)$ denotes the non-linear activation function. For $s = 1$, $a_i^{(1)}$ is equivalent to the i^{th} input feature. $W_{ij}^{(s)}$ specifies the weight associated with the connection between unit j in layer s and unit i in layer $s+1$, and $b_i^{(s)}$ indicates the bias associated with unit i in layer $s+1$. In the backward propagation step, a gradient descent based algorithm is applied to adjust (fine-tune) network parameters. The training procedure continues until a local optima is reached via the deep network.

DL Training Phase. Alg.2 provides the pseudocode of TinyDL training phase. TinyDL requires only one pass through each arriving sample to update the coefficient matrix C. In Alg.2, n_{batch} is a user-defined variable that controls the frequency of updating the classifier based on the arriving rate of

training samples. After every n_{batch} new training data, TinyDL updates the acquired DL model through multiple rounds of forward and backward propagation using the expanded matrix C that contains both the new and previously computed vectors.

The inputs of the DNN module include projected training samples (matrix C), the corresponding label for the training samples (L_T), current network parameters $param = (weights,\ biases)$, and the number of units per layer $layer_{size}$. $layer_{size}$ is a vector of integers whose first component should be equal to l (number of features in the block-sparse matrix C), and its last element indicates number of classes according to the application. Each number in between the first and last components of the $layer_{size}$ indicates the number of units per hidden layer in the chosen DNN topology.

Algorithm 2 : TinyDL (Training Phase)

Inputs: Measurement matrix (A), Transformation parameters (α, k, l), Sample size (n_{epoch}), Training labels (L_T), and DL topology ($layer_{size}$)
Output: Dictionary matrix D, coefficient matrix C, and DL parameters $param$.

$D \leftarrow$ empty
$param \leftarrow$ empty
$j \leftarrow 0$
$i \leftarrow 0$
while *(true)* **do**
  $C_i \leftarrow 0$
  if D is empty **then**
    $W_p(a_i) = 1$
  else
    $W_p(a_i) = \frac{\|D(D^tD)^{-1}D^ta_i - a_i\|_2}{\|a_i\|_2}$
  end if
  if $W_p(a_i) > \alpha$ **and** $j < l$ **then**
    $D \leftarrow [D, a_i/\sqrt{\|a_i\|_2}]$
    $C_{ij} = \sqrt{\|a_i\|_2}$
    $j \leftarrow j + 1$
  else
    $C_i \leftarrow OMP(D,\ a_i,\ k)$
  end if
  $C \leftarrow [C, C_i]$
  $i \leftarrow i + 1$
  if $i\ mod\ n_{batch} == 0$ **then**
    $param \leftarrow DNN(C,\ param,\ layer_{size},\ L_T)$
  end if
end while

DL Execution Phase. Once the classifier is trained using Alg.2, there are two main steps to predict the class label for each data measurement in the execution phase: (i) Each test sample is projected based on the learned dictionary matrix D. (ii) The corresponding coefficient vector C_i is fed into the trained DL-based classifier to obtain the class label. The execution phase only requires a forward propagation (Eq.2) for each transformed data sample; thereby the execution latency is proportional to the number of units per layer ($layer_{size}$).

C. TinyDL Automated Customization

Our customization module aims to find an optimized set of transformation parameters such that the DL training and execution costs are significantly reduced while the inference accuracy is minimally affected.

Memory Customization. The dictionary size indicated by l is equal to the feature space size of the coefficient matrix C (the input layer size of the DL network) and has a direct effect on the memory storage and overall DL network size. The memory footprint of the DL network is defined by $(ml + ln + size(param))$; l, m, and n are data-specific variables while $size(param)$ depends on the DL topology (e.g., number of edges and units per layer). TinyDL adaptively tunes its dictionary size l with respect to the input feature space size m, and the available memory bandwidth obtained by platform profiling. Note that the dictionary matrix $D_{m\times l}$ is constructed from carefully sampling columns of the data matrix $A_{m\times n}$. Thus, the column space of D is contained in the column space of A, which in turn implies that $rank(DD^+A) = rank(D) \leq l \leq m$. Here, D^+ denotes the pseudo-inverse of matrix D. This guarantees that the dictionary matrix D at most consists of m samples.

Error Customization. The sparsity level k has a significant impact on the ultimate inference accuracy and pre-processing overhead. Many contemporary large datasets can be represented by a composition (ensemble) of several lower-dimensional subspaces. This composition of data can be effectively modeled by using a small subset of data as shown in [2], [3], [4]. TinyDL uses a small subset of data (e.g., 5%) to optimize its pre-processing parameters (e.g., k) with respect to different applications and resources provisioning.

Runtime Customization. Transformation overhead of each newly arriving sample is a deterministic function of dictionary size l, sparsity level k, and the input feature space size m. TinyDL takes the user-defined runtime budget into consideration and tunes the algorithmic parameters accordingly. The runtime budget could be dictated either by the arriving rate of sensor measurements or the buffer size for storing incoming samples in the target platform.

III. Experiments

All our design experiments are carried out on a constrained embedded system named Jetson TK1. NVIDIA Jetson TK1 is a full-featured platform for realizing computer vision, robotics, security, automotive, and mobile sensing embedded applications [5]. It includes 192 CUDA cores and 4-Plus-1 quad-core ARM Cortex A15 CPU with 2 GB memory. In our evaluations, data transformation is performed by standard Message Passing Interface (MPI) system using the 4 quad-core ARM processors and the DL training and execution have been performed on the available CUDA cores. We use TinyDL to realize three different contemporary learning applications: (i) Smart sensing [6], (ii) Indoor localization [7], and (iii) Speech recognition [8].

To evaluate TinyDL performance in presence of noisy samples, we perform three experiments using the smart-sensing data with (i) No additive noise, (ii) Signal to Noise Ratio (SNR) $= 10$, and (iii) SNR $= 15$. In these experiment, we use $k = 20$, and $l = 100$ for signal transformation, while each original input sample has 5625 features. Figure 2 shows test error over time for three different input SNRs. As shown, TinyDL is not sensitive to noisy inputs (less than 2% variation in final inference accuracy) which makes it a suitable choice for development of sensing inferences in presence of noisy sensor measurements.

The graphs in Figure 2 illustrate that our data transformation greatly speedups the DL training process, while it maintains a competitive inference accuracy. This increase in the convergence rate is mainly due to the block-sparse structure of matrix C that facilitates deep learning by effectively representing data

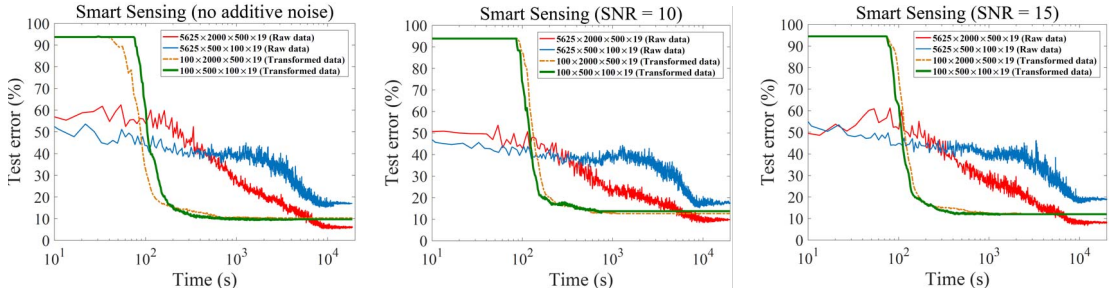

Fig. 2: Inference error over time for different input SNRs. The sparsity level for transformed data is set to be $k = 20$, and the dictionary size is $l = 100$. The reported time correspond to TinyDL runtime on the GPU of Jetson TK1 board.

as an ensemble of lower-dimensional subspaces. As the dictionary evolves over time, it better captures dynamic structure of data, and subsequently increases the error reduction rate in TinyDL. In addition, our data transformation also enables us to achieve the same level of accuracy by utilizing smaller number of units per layer compared to the conventional scenario where the *raw* data is fed to the DL network. This reduction in the overall DL network size translates to a lessening in resource utilization as well as training and execution latency.

Table I elaborates the effectiveness of TinyDL in realizing various DL-based robotic and sensing applications on a resource-constrained device. To boost the inference accuracy and avoid over-fitting, we employ dropout technique [9] in our baseline implementation. Stochastic gradient descent with momentum [10] is used for back-propagation. We report performance improvement achieved as a result of TinyDL pre-processing compared to the conventional approach where raw data features are used of DL training.

TABLE I: Performance improvement achieved by TinyDL over conventional deep learning approach.

Application	Pre-processing Overhead	Training Energy	Training Runtime	Execution Runtime	DL Network Size Reduction
Smart Sensing	$0.81\ ms/sample$	99.6×	20.9×	108.3×	120.3×
Indoor Localization	$0.09\ ms/sample$	5.6×	2.7×	19.4×	19.5×
Speech Recognition	$0.56\ ms/sample$	4.2×	2.3×	6.2×	7.3×

IV. Related Work

There are some early examples of DL being applied in mobile settings including the speech recognition models used by phones today [11]. Such models, however, operate entirely off-device in the cloud and have not exploited the computational resources available on current mobile devices. A recent work [12] investigates the potential of using DL as an alternative approach for commonly used learning models such as GMM, SVM, or DT in mobile sensing applications. This work demonstrates that execution of DNN can incur resource overhead close to the simplest comparison models such as DT (which does not have a high accuracy), yet simultaneously have accuracy levels equal to the best tested alternatives (e.g., GMM, or SVM). However, even [12] leaves training of such deep networks for the clouds and only focuses on local usage of the trained DNNs for classification.

In our recent work [3], [13], [14], we corroborate how the use of resource-aware data transformation as a pre-processing step can be leveraged to customize DL training and execution workload in accordance to the underling resource provisioning. TinyDL is developed based on a novel extension of our proposed method in [3], [13], [14]. In particular, TinyDL further reduces the complexity of signal pre-processing by greedily select the best fitting dictionary samples to project each newly arrived data while delivering the same accuracy.

V. Conclusion

This paper presents TinyDL, a novel end-to-end framework for realization of sensing and understanding tasks on resource-constrained platforms using DL models. TinyDL adaptively learns and customizes the hybrid structure of the streaming input data to improve system performance in terms of memory, power, energy consumption, and training runtime. We evaluate TinyDL on three contemporary sensing applications including smart sensing, indoor localization, and speech recognition. Our experiments demonstrate up to 100-fold improvement in comparison with the best known prior solutions, none of which were implemented on a mobile platform due to the demanding computational overhead associated with DL methods.

Acknowledgments. This work was supported in parts by the ONR (N00014-11-1-0885), NSF TrustHub (1649423), and NSF SRC (1619261) grants.

References

[1] L. Deng and D. Yu, "Deep learning: methods and applications," *Foundations and Trends in Signal Processing*, 2014.

[2] B. D. Rouhani, E. M. Songhori, A. Mirhoseini, and F. Koushanfar, "Ssketch: An automated framework for streaming sketch-based analysis of big data on fpga", FCCM, 2015.

[3] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Delight: Adding energy dimension to deep neural networks", ISLPED, 2016.

[4] A. Mirhoseini, B. Rouhani, E. Songhori, and F. Koushanfar, "Perform-ml: Performance optimized machine learning by platform and content aware customization", DAC, 2016.

[5] https://developer.nvidia.com/jetson-tk1, "Jetson tk1", 2015.

[6] https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities, "UCI repository", 2015.

[7] https://archive.ics.uci.edu/ml/datasets/UJIIndoorLoc, "UCI repository", 2015.

[8] https://archive.ics.uci.edu/ml/datasets/isolet, "UCI repository", 2015.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", JMLR, 2014.

[10] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning", ICML, 2013.

[11] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer *et al.*, "Recent advances in deep learning for speech research at microsoft", ICASSP, 2013.

[12] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?", HotMobile, 2015.

[13] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Deep^3: Leveraging three levels of parallelism for efficient deep learning", DAC, 2017.

[14] ——, "Going deeper than deep learning for massive data analytics under physical constraints", CODES+ISSS, 2016.