electronics

MDPI

Article

Revisiting Resource Management for Deep Learning Framework

Erci Xu * and Shanshan Li

School of Computer Science, National University of Defense Technology, Changsha 410073, China; shanshanli@nudt.edu.cn

* Correspondence: xuerci@nudt.edu.cn; Tel.: +86-18-6707-91940

Received: 16 February 2019; Accepted: 6 March 2019; Published: 16 March 2019

Abstract: The recent adoption of deep learning for diverse applications has required infrastructures to be scaled horizontally and hybrid configured vertically. As a result, efficient resource management for distributed deep learning (DDL) frameworks is becoming increasingly important. However, existing techniques for scaling DDL applications rely on general-purpose resource managers originally designed for data intensive applications. In contrast, DDL applications present unique challenges for resource management as compared to traditional big data frameworks, such as a different master–slave communication paradigm, deeper ML models that are more computationally and network bounded than I/O, the use of heterogeneous resources (e.g., GPUs, TPUs) and the variable memory requirement. In addition, most DDL frameworks require data scientists to manually configure the task placement and resource assignment to execute DDL models. In this paper, we present Dike, an automatic resource management framework that transparently makes scheduling decisions for placement and resource assignment to DDL workers and parameter servers, based on the unique characteristics of the DDL model (number and type of parameters and neural network layers), node heterogeneity (CPU/GPU ratios), and input dataset. We implemented Dike as a resource manager for DDL jobs in Tensorflow on top of Apache Mesos. We show that Dike significantly outperformed both manual and static assignment of resource offers to Tensorflow tasks, and achieved at least 95% of the optimal throughput for different DDL models such as ResNet and Inception.

Keywords: deep learning; resource management; cloud computing

1. Introduction

Today, distributed deep learning (DDL) is widely used in different areas ranging from image classification to speech recognition [1,2]. Various open source frameworks for DDL such as Tensorflow, MXnet, and Azure Machine Learning [3,4] are being offered as services by cloud providers or deployed by users in private clusters using resource containers. As a result, efficient resource management [5,6] for DDL frameworks is critically important.

Resource management in major DDL frameworks is still evolving and does not account for the unique characteristics of the machine learning jobs. For example, the data scientist has to address the following four questions while deploying the DDL model: (1) How many DDL tasks need to be launched? (2) How much resource allocated for each task? (3) What is the role or functionality of each task? (4) Which physical node should be used for launching each task? As these questions are specific to the requirements of the DDL model and have multiple possible answers, users need to iteratively try out different deployment plans, which requires considerable manual tuning. Moreover, even if a good solution is obtained, it may be no longer suitable if users to change the cluster configurations, use different models, or train on another dataset.

Previously, answering similar questions in other data-intensive analytic frameworks [7,8] is often straightforward due to a clear requirement to minimize I/O to disk and network transfers [9]. As a result, most big data frameworks provide resource assignment by allocating sufficient memory to each task and reduce frequent disk I/O (addressing Questions 1 and 2). For task placement, locality has been the foremost priority to minimize network transfers (addressing Questions 3 and 4).

Unfortunately, simply applying these principles to resource management for DDL frameworks would not always lead to a good solution. This is attributed to three unique features of DDL: deeper ML model pipelines that are both more computationally and network-bound than I/O, master–slave communication paradigm between parameter-servers and workers, and the use of heterogenous resources for DDL such as GPUs and variable memory.

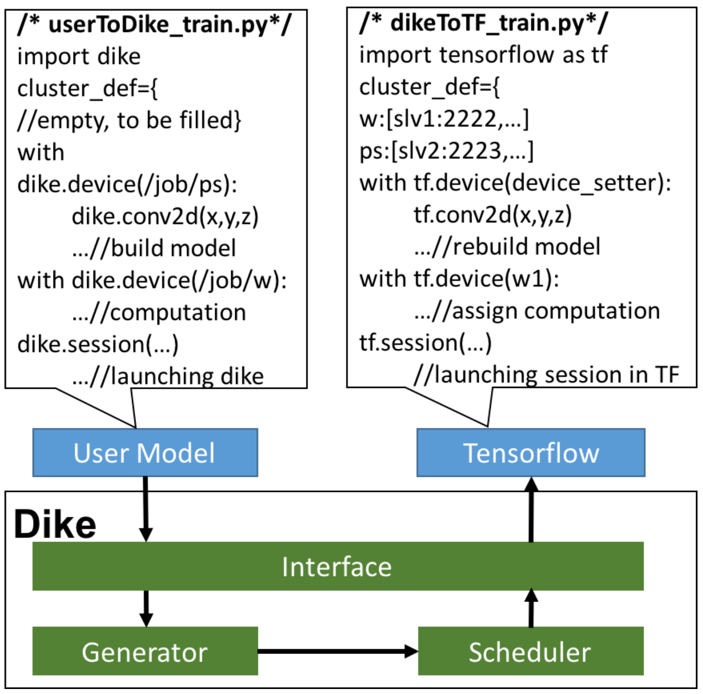
First, most DDL models use a computation-intensive procedure called gradient descent. Iteratively computing to minimize the loss function in gradient descent is the key bottleneck rather than the I/O [10]. Similarly, the DL pipelines usually comprise of several layers of highly computationally-intensive operations and backward propagation of parameters. As a result, allocating resources to tasks in DDL based on memory requirements alone may lead to inefficient memory utilization and waste of computing resources.

Second, DDL introduces a new role for classic distributed master–slave communication parameter server (PS) [11]. This affects the task placement as a slave node can either serve as a worker executing the DL models or a PS that maintains a synchronized view of the DL model parameters across different workers. Accordingly, resources assigned to a PS have very different requirements than that of a worker. Similarly, a PS needs to be placed in a location so as to minimize the network overhead for synchronizing the parameters with the workers for which it is responsible.

Third, DDL clusters are usually equipped with very heterogenous hardware such as different compute power (GPU and TPU) [12], and different provisioned memory [3]. As a result, selecting the right compute and memory offers to provision for the different DL model operators becomes critical. For example, GPUs offer high parallelism and concurrency levels well suited for computation on workers. Most data analytic frameworks run on homogeneous configurations. However, simply binding the GPU nodes to workers and CPUs to PS, may not result in the optimal placement, as different workers need to be co-located with different PS.

In existing systems [3,4], most of the resource assignment and task placement is still done by the data scientist by selecting the physical nodes or resource containers for executing different workers and PS. For example, Tensorflow requires the data scientists writing the DL model to determine the total number of workers and PS, location of PS/worker tasks, and their CPU/GPU and memory configurations. In contrast, resource management frameworks based on Apache Mesos [13,14] use coarse-grained levels for resource assignment and scheduling strategies not aware of the DL model characteristics. In addition, data scientists still need to determine the total number of tasks and resources for each of them. As a result, we find that their task assignment is not optimal in most cases.

In this paper, we present Dike, an online resource management framework for DDL, as shown in Figure 1. Data scientists only need to write their DL models, which are similar to the Tensorflow models, however, without requiring any resource management details. Dike creates a new wrapper Context over Tensorflow context to intercept and capture runtime information, including model details and cluster configuration, from the DDL framework. Dike then passes this information to its Generator component to produce multiple candidate resource assignment plans. These plans are finally evaluated by Dike's Scheduler based on a cost model, which decides the best plan and its placement for each task. We show that Dike achieves at least 95% of the optimal performance for distributed DDL workloads and automates most of the cluster resource management.



```
/* userToDike_train.py*/
import dike
cluster_def={
//empty, to be filled}
with
dike.device(/job/ps):
        dike.conv2d(x,y,z)
        ...//build model
with dike.device(/job/w):
        ...//computation
dike.session(...)
        ...//launching dike
```

```
/* dikeToTF_train.py*/
import tensorflow as tf
cluster_def={
w:[slv1:2222,...]
ps:[slv2:2223,...]
with tf.device(device_setter):
        tf.conv2d(x,y,z)
        ...//rebuild model
with tf.device(w1):
        ...//assign computation
tf.session(...)
        //launching session in TF
```

Figure 1. Architecture of Dike.

The rest of the paper is organized as follows: Section 2 introduces resource management and its impact on performance for DDL. Sections 3 and 4 describe the design and evaluation of different components of Dike.

2. Background

In general, resource management has two responsibilities, resource assignment and task placement [13,15]. Resource assignment determines the total numbers of tasks and the resource to be allocated for each task. For example, a Spark application may determine the total number of tasks and resource binding based on available memory and the size of data input. Task placement includes deciding location for each task to be launched. For instance, Delay Scheduling algorithm [16] in Hadoop tends to assign tasks based on data locality. Currently, both parts are mainly handled by manual effort in DDL due to performance concern and lacking of competitive tools.

To further understand resource management process of deploying DDL, we use the following example, as shown in Figure 2, which describes typical procedures of a Tensorflow deployment on a Mesos cluster. Mesos first receives cluster configuration from all slaves in Step (1). Mesos master then offers resources to a ready-to-launch application, for instance a Tensorflow model. In Step (3), Tensorflow, after receiving the resource offers from Mesos, determines specific resource and orchestrates the placement for each task. Finally, the tasks are sent from master to each instance, either a container or a physical node, to launch the training session, as shown in Step (4). The DDL training session can be regarded as a batch processing job where, in each batch, the workers download the parameters that are shared among the parameter servers, and perform forward and backward propagation. Finally, the workers update the changes to the corresponding parameter servers.

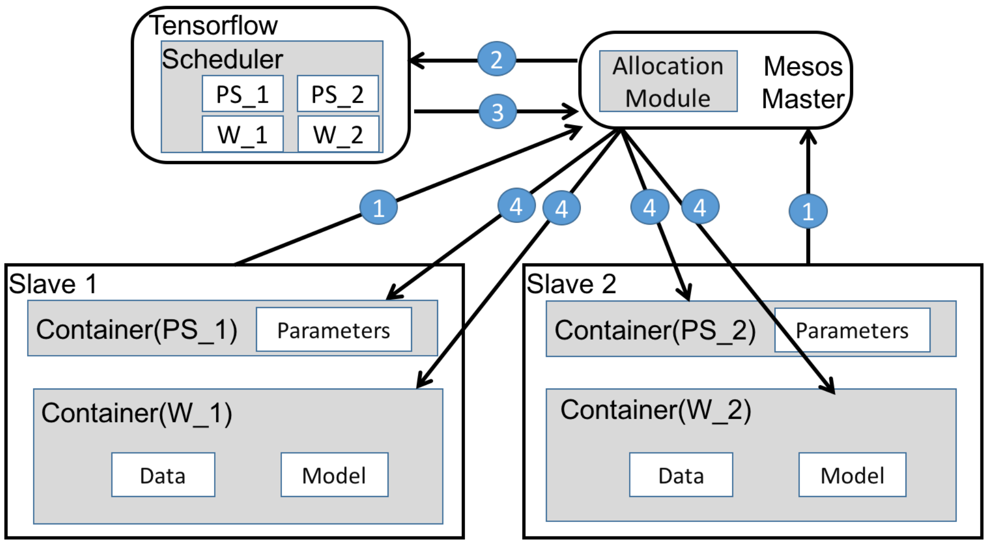


Figure 2. Procedures of resource management in Mesos for a Tensorflow application.

As shown in the procedure, resource management is the key to the overall throughput of DDL. An incorrect decision can easily slow down the training. For example, an insufficient number of PS or Worker tasks can lead to resource underutilization; an inefficient task placement can cause congestion in the network; a biased resource assignment may starve the GPUs due to waiting for CPUs to process inputs. In addition, due to the three unique features mentioned above, simply adopting the observations and guideline from the data-intensive frameworks may not yield optimal performance. The evaluation in Section 4 further reveals the impacts of unideal resource management.

3. Design

The design goal of Dike is to free the deep learning frameworks users from considering various system configuration related tasks. Therefore, the first design principle is to require relative less changes to the current programs. The second principle is to provide high performance. As shown in Figure 1, the methodology of Dike is as follows: (1) Users transfer the tasks resource assignment and tasks placement to Dike in the code. (2) The Dike intercepts the program via embedded APIs and dynamically chooses the best options for resource assignment and task placement through algorithms. (3) Dike invokes original deep learning framework with an updated program with detailed assignment of resources and task placement and start the training.

In details, as shown in Figure 1, Dike consists of three major components: Interface to obtain runtime information and launch a DDL session with the optimal plan; Generator to generate resource assignment options; and Scheduler to decide best option and its corresponding task placement. To clearly illustrate our design, we demonstrate Dike according to execution timeline in a detailed example, as shown in Figure 3. At each stage, we describe the input, the functionality and the output. Overall, for Dike, the initial inputs are the client program and the cluster configuration. The final outputs are total numbers of tasks, location of each PS/Worker, and resource of each PS/Worker.

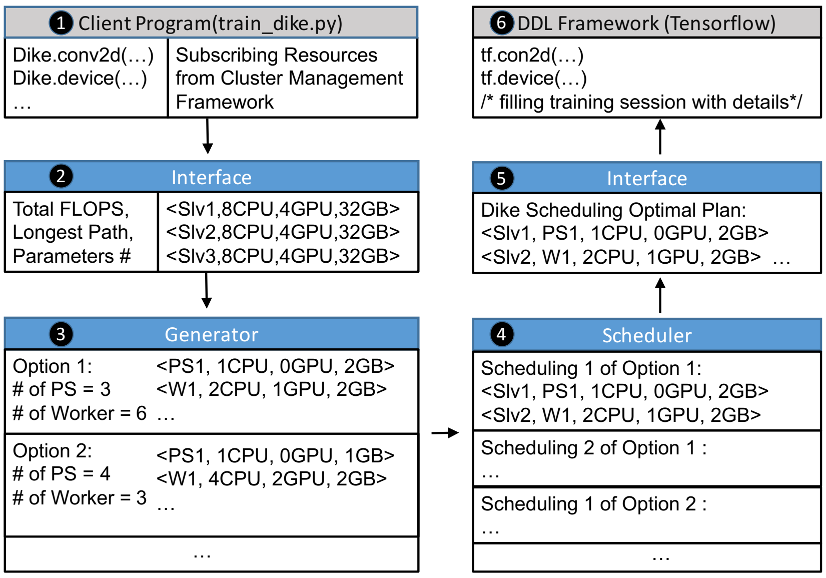

1 Client Program(train_dike.py)
Dike.conv2d(…)
Dike.device(…)
…
Subscribing Resources from Cluster Management Framework
2 Interface
Total FLOPS, Longest Path, Parameters #
<Slv1,8CPU,4GPU,32GB>
<Slv2,8CPU,4GPU,32GB>
<Slv3,8CPU,4GPU,32GB>
3 Generator
Option 1: # of PS = 3 # of Worker = 6
<PS1, 1CPU, 0GPU, 2GB>
<W1, 2CPU, 1GPU, 2GB>
…
Option 2: # of PS = 4 # of Worker = 3
<PS1, 1CPU, 0GPU, 1GB>
<W1, 4CPU, 2GPU, 2GB>
…
4 Scheduler
Scheduling 1 of Option 1:
<Slv1, PS1, 1CPU, 0GPU, 2GB>
<Slv2, W1, 2CPU, 1GPU, 2GB>
Scheduling 2 of Option 1 :
Scheduling 1 of Option 2 :
5 Interface
Dike Scheduling Optimal Plan:
<Slv1, PS1, 1CPU, 0GPU, 2GB>
<Slv2, W1, 2CPU, 1GPU, 2GB> …
6 DDL Framework (Tensorflow)
tf.con2d(…)
tf.device(…)
/* filling training session with details*/

Figure 3. Detailed process of Dike.

3.1. Interface

Dike's Interface has two main responsibilities: retrieving runtime information (Figure 3, Step 2) and executing the optimal plan by launching a DDL session with resource management details (Figure 3, Step 5). Specifically, in Step 2, users will utilize Dike's APIs to build the model as a client program. By running this program, we can capture model and training session details, including model architecture, layer types and batch size. In addition, this program would be registered on Mesos to subscribe currently available resources. These two pieces of information will be processed before feeding to Generator and Scheduler to generate an optimal resource management plan. In Step 5, after receiving an optimal plan from the Scheduler, Interface will start a Tensorflow session and fill previously user-defined model with resource assignment and task placement according to the received plan. From users' perspective, they only need to build the model and all the resource management details are handled by Dike.

To help users to use Dike and avoid reinventing the wheels, we designed Interface APIs as wrappers to original Tensorflow ones. Each wrapper has the same parameter list but the name would be changed accordingly, such as from $tf.conv2d(i,j,k)$ to $dike.conv2d(i,j,k)$. These APIs will first report details to Dike and wait for the optimal plan. Once received, these wrappers will return as the original function call but certain parameters are filled by Dike to carry out the optimal plan. Thus, when switching to Dike, users only need to import Dike module and change function names in their model. There are three types APIs we modified: Model related APIs, for estimating resource requirement; Session APIs, for controlling session life cycle; and Pacement APIs, for Dike to execute optimal plan.

As not all model details are related to resource management, Interface would further process the information to generate three intermediate results as outputs. First, the amount of parameters is used to estimate memory requirement and network traffic because each worker needs to download and keep a copy of all parameters in each batch. Secondly, the total amount of float point computation (FLOPs) in a model can be obtained by calculating the number of parameters that are involved in gradient descent. Acquiring FLOPs enables us to estimate computation time which helps Dike to find a balanced CPU-to-GPU ratio in resource assignment and avoid resource underutilization or GPU starving. Please note that different algorithms can considerably change the FLOPs and execution time. Our prototype currently assumes users always use one algorithm the whole time Thirdly, the longest path of the model is used to accurately estimate computation time of complex models such as Inception-V3 [1]. As latest models can have branches and dropouts [17], it can be difficult for

Dike to simply rely on FLOPs to estimate computation time. Longest path, in this case, provides an adjustment to the estimation of time needed for computation.

3.2. Generator

From Interface, Dike receives the amount of parameters, the FLOPS of the model, the longest path of model and the cluster configurations. With these, Generator can generate possible resource assignments, which includes two parts, the numbers of tasks and the resource on each task, namely the numbers of CPU cores, numbers of GPU devices, memory size and storage space.

A naive approach of generating such assignments is to explore all the possibilities in a brute-force manner. Apparently, the search space is of exponential complexity. Specifically, a five-node cluster where each node has quad-core cpu, 4 GPUs and 16 GB memory can easily generate more than 41 million different combinations. To offer Dike as an online solution, based on our observations and experiences in deployment, we designed the following pruning techniques to minimize the search space:

First, workers need to be of same resources assignment. DDL in fact is a batch processing job. Binding workers with different resources is intentionally creating stragglers. However, resource assignment of PS servers, especially memory, should be decided by how many parameters to be stored within and hence can be different from one to another.

The second pruning technique is to utilize FLOPS to determine a range for CPU-to-GPU ratio. Although workers mainly using GPUs to perform training, CPU is still needed for network communication and unpacking data from storage. The ideal situation is that data are prepared by CPU and processed by GPU seamlessly. Therefore, Dike uses the FLOPs and hardware computation information to strike a balance between CPU and GPU. Furthermore, if the model is with branches, the Dike will use the longest-path FLOPs over total FLOPs as an adjustment.

The third one is to limit the numbers of tasks. Thanks to resource segregation and containers, a physical node can launch multiple instances. Based on our observations in deploying DDL, we limit the total number of PS/Worker as follows. The range for PS is between one and the number of physical machines as there is no point to have more than one PS tasks on the same node. The range of worker is between number of GPU-equipped nodes to number of GPU devices as each worker needs to at least has one GPU device and a worker crossing two nodes is generally not a favorable option.

Thanks to these pruning techniques, deploying a DDL model on a five-node cluster in Dike usually yields fewer than 20 unique resource assignment options.

3.3. Scheduler

After receiving multiple resource assignment options from Generator, Scheduler will determine a best task placement for each option. Then, Dike would compare each opted task placement across all options and select the optimal plan to return to the Interface. This optimal plan contains both resource assignment and placement for each task.

Dike determines and compares different task placement by using a cost model. The goal of this model is to minimize network traffic which is mostly generated by workers downloading parameters. The approach is to run a dynamical programming model that tries to place tasks into several containers. We formulate the scheduling as a problem similar to bin-packing, which can be formally described as follows:

$$\sum_{k=1}^{n}\sum_{j=1}^{m}\{Mem_j \times Conn_{jk} \times Cost_{in} + Mem_j \times (1 - Conn_{jk}) \times Cost_{out}\} \tag{1}$$

Subject to following conditions:

$$\sum_{k=1}^{n}\sum_{j=1}^{m}\sum_{r}(R_{jr} \times P_{ij} + R_{kr} \times W_{ik}) \leq B_{ir} \tag{2}$$

$$Conn_{jk}, P_{ij}, W_{jk} \in \{0, 1\} \tag{3}$$

$$r \in \{CPU, GPU, Mem, Disk\} \tag{4}$$

$$k \in \{1, ..., n\} \text{ and } \sum_{k=1}^{n} W_{ik} = 1 \tag{5}$$

$$j \in \{1, ..., m\} \text{ and } \sum_{j=1}^{m} P_{ij} = 1 \tag{6}$$

In the equations, we mark r as the resource type; and k and j as the index for worker and PS, respectively. Consequently, R_{jr} and R_{kr} indicate the r type of resource requirement for jth PS or kth Worker. We mark each node as a bin B_i and its capacity of a certain type of resource r is denoted as B_{ir}. We mark the presence of a task in a bin as W_{ik} or P_{jk} where 1 means place this task in bin B_i and 0 otherwise. Finally, $Conn_{jk}$ indicates whether PS task j and Worker task k are co-located in the same physical machine. $Cost_{in}$ and $Cost_{out}$ describe the weighted cost of communication cost within or cross node(s). The costs can be further specified as $Cost_{out\_jk}$, which is cross node cost from j to k on a peer-to-peer basis.

Our goal is Equation (1) and subjects to a series of conditions including: Equation (2) limits the maximum tasks to be placed on one bin; Equations (3) and (4) limits the range; and Equations (5) and (6) indicates each task should be placed into one bin. The outcome of Equation (1) is a *Cost* and our goal is to find the smallest *Cost* that satisfies all conditions. The smallest cost indicates the optimal assignment solution for that workload.

It is well known that bin-packing is an NP-hard problem and hence requires exponential time complexity. Fortunately, with previous pruning and optimizations, our tasks generations only produce a limited number of options. According to our simulation, deploying 300 nodes cluster consumed up to 23 min in Dike's resource management, which was significantly less than iterative manual effort.

After a minimum goal is reached, we can translate the equation into task placement and use the corresponding R_{jr}, R_{kr}, P_{ij}, and W_{ik} as resource assignment, which would be sent back to the Interface to launch a DDL session as described in the previous subsection.

4. Evaluation

Our experiments tested Dike against three resource management approaches with different setups in scale, resource heterogeneity, dataset and models. The results show Dike delivered optimal performance in all small scale tests and at least 95% at larger scale deployment with at most 19% iterative approach time consumption, which provides a strong portability.

4.1. Methodology and Environment

Our evaluation compared Dike against Static Assignment with Round-robin placement (SAR_Basic), a fine-tuned version of SAR (SAR_Tuned) and iterative manual approach on training throughput. As shown in the breaking-down chart in Table 1, each has a different proportion of manual involvement. SAR_Basic decides the resource assignment based on manually estimating the available resources and SAR_Tuned improves performance by asking users to repeatedly tune the resource assignment, which in return consumes more time. Both SAR methods place tasks in a round-robin manner, a common practice in current DDL frameworks [3,4]. The iterative approach yields the best performance as it exhaustively tries out all possible combinations of resources assignment and placement possibilities. Intuitively, it consumes the largest amount of time.

Table 1. Overview of different resource management.

Scheduling Approach	Resource Estimation	Resource Assignment	Task Placement
Dike	Auto	Auto	Auto
SAR_Basic	Manual	Auto	Auto
SAR_Tuned	Manual	Manual	Auto
Iterative	Manual	Manual	Manual

The test environment is a five-node cluster where each node is equipped with 4 Nvidia K80 GPUs, 8-core Intel i7-6900K CPU, 32 GB memory and is interconnected by 10 Gbps network. Regarding models, we choose Resnet [18] and Inception-v3 [1] as they are of different architectures and the number of parameters can be readily changed. By comparing the two varieties of Resnet (i.e., Resnet 50 and Resnet 152), we could quantitatively measure the Dike's performance when changing the number of parameters and the amount of computation in similar models. By comparing Resnet model against Inception-v3 model, we could examine whether Dike is capable of dealing with modern complicated models that have been used in industry environments. For input, we selected Cifar10 [19] (around 170 MB) and Imagenet [20] (around 55 GB), which include situations where a single node both can and cannot cache the complete dataset in its memory. Cluster configuration had two sub-dimensions, scale and heterogeneousness. Our experiments ranged from 5 to 20 instances, among which we achieved heterogeneousness by muting GPUs in certain instances.

4.2. Performance

To cover different input, models, scales and cluster configurations, we conducted three sets of experiments and set throughput of manual iterative approach as the benchmark for optimal performance.

The first set of experiments focused on model difference. As shown in Figure 4A, we tested Resnet 50, Resnet 152 and Inception-v3, which approximately have 21.3 million, 64.0 million and 21.8 million parameters. The tests were run on Cifar10 dataset with 5, 10 and 20 instances. The experiment showed that Dike always achieved optimal (100%) performance on Resnet models on 5 and 10 instances. For larger scales, such as Inception-v3 with 20 instances, Dike achieved 95% of the optimal performance. The gap was caused by different placement of PS servers as Dike ran the scheduling algorithm based on profiling peer-to-peer cost, which could be slightly different in practice. The main setbacks of two SAR approaches were different: SAR_Basic ignored the differences in resource requirement between workers and slaves; and SAR_Tuned ignored co-existence of PS/worker and therefore misplaced the task locations, which further caused resource underutilization and network congestion.

The second set of experiments, as illustrated in Figure 4B, was to examine performance under various input. The results indicate Dike's performance was not affected by the size of input because, instead of locality-oriented resource management, Dike assigned resource based on computation requirement and dynamically determined the task location based on network traffic. SAR approaches, however, were severely affected by the memory-first resource assignment strategy. For smaller datasets, such as Cifar10, the locality became irrelevant, as each worker could have a complete copy of the dataset. As a result, task placement regressed to random round-robin placement. Larger datasets, such as Imagenet, affected the resource assignment of PS because memory were allocated to cache input rather than storing parameters.

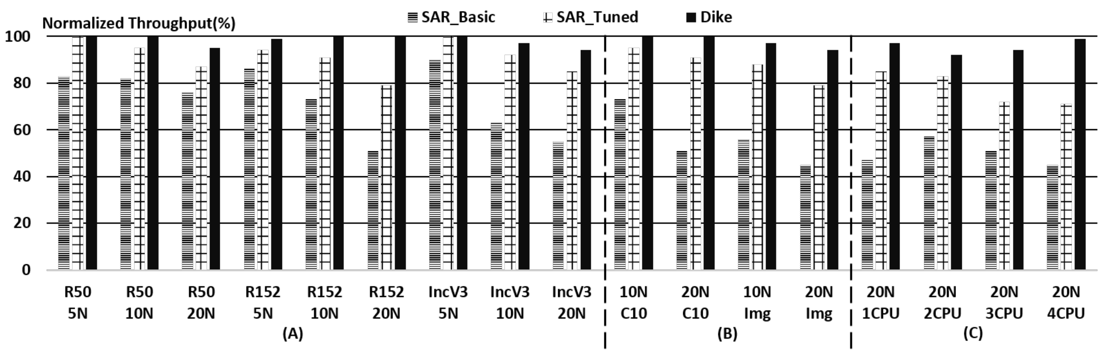


Figure 4. Performance comparison on different models/scale/heterogeneity. The X-axis is the different experiment setups where R, $IncV3$, N, $C10$, and Img are short for Resnet, Inception-V3, Node, Cifar10, and Imagenetl respectively. Y-axis is the normalized throughput. The value 100% indicates same performance as iterative approach.

The third set of experiments was conducted in a heterogeneous environment, as shown in Figure 4C. We tested four different types of instances where CPUs-only instances increased from 1 to 4. The results reveal that Dike adapted well with cluster configuration by generating the right amount of PS tasks and accommodated them in the CPU-only instances. SAR approaches, even after manually assigning the right amount of tasks, could still suffer from misplacing the tasks, as they were unaware of heterogeneous resource environment.

5. Related Work

DDL framework Optimization: Most deep learning frameworks, such as MxNet and Tensorflow, support manual setups for resource and task assignment. As an example, in the Tensorflow official tutorial for distributed training [21], when users to need deploy a job among different GPU devices, users typically need to assign a certain task to a specific device, which is called "Manual Setting". Recent development provide APIs for developers to orchestrate task placement and resource assignment. For example, latest releases of Tensorflow enable users to automatically place tasks among parameter servers in a load-balancing manner on a Kubernetes [6] cluster. In addition, third-party frameworks, such tfmesos [14], help users place tasks in customized manners. However, those improvement usually only tackle one side of the problem, either resource assignment or task placement, as shown in Table 2.

Table 2. Status quo of current deep learning frameworks.

Techniques	Platform	Resource Assignment	Task Assignment
TF Load Balancing	Kubernetes	Auto	Manual
tfmesos	Mesos	Manual	Auto
MxNet	ssh (Manual)	Manual	Manual
Continuum	ssh (Manual)	Manual	Manual
Dike	Mesos	Auto	Auto

HPC Resource Management: In many ways, DDL workloads resemble classic HPC applications, such as computation bound, heterogeneous resource and long running time. Nonetheless, certain barriers make it infeasible to directly utilize resource management tools from HPC. For instance, HPC cluster is usually equipped with specialized hardware, which requires the resource management to perform corresponding optimization for a limited set of applications. DDL users, on the other hand, desire deploying workloads in consumer-class hardware environment, such as AWS cloud, with generic resource managing tools such as Apache Mesos [13].

Scaling DDL workloads: AWS auto-scaling enables DDL workload to be executed in different scales and helps users managing cloud resource. However, simply scaling the numbers of PS and worker does not yield best performance, as it is also affected by network traffic, node location and resource per node.

6. Conclusions and Future Work

Recently, wide adoptions of distributed deep learning deployment have proposed new challenges to resource management. New communication paradigm, computation/network bounded workloads and heterogeneous resources make it infeasible to apply previous principles and techniques. As a result, users, such as data scientists, need to spend considerable amount of time in tuning resource management for better performance. In this paper, we present Dike, an automatic online resource management tool for deploying DDL workloads. Our experiments indicated our prototype could deliver ideal performance in small-scale testing and up to 95% of the optimal performance (obtained by exhaustive procedures) in larger scale deployment. The evaluation proved that Dike could automate the time-consuming and often-manual procedures of resource assignment and task placement in deploying deep learning workloads.

Future work of Dike includes: supporting various DDL frameworks, e.g., Mxnet and CNTK; adapting to more types of models, such as LSTM [2] and Attention [22]; and connecting to different backends. Ideally, we aim at offering Dike as a universal middleware between DDL frameworks and cluster resources, where users in the future only need to be concerned with building the model while Dike handles all resource management details.

Author Contributions: The first author (E.X.) was responsible for the project administration, programming, and writing the original draft; and S.L. was responsible for the handling the funding/resources and revising the manuscript.

Funding: This research was funded by NSFC Nos. 61872373 and 61872375.

Acknowledgments: The authors in this paper would like to thank the anonymous reviewers for their feedbacks and our editor Zackery Wang for help processing the paper.

Conflicts of Interest: The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the inception architecture for computer vision. *arXiv* **2015**, arXiv:1512.00567.
2. Sutskever, I.; Vinyals, O.; Le, Q.V. Sequence to sequence learning with neural networks. *arXiv* **2014**, arXiv:1409.3215.
3. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; USENIX Association: Berkeley, CA, USA; pp. 265–283.
4. Chen, T.; Li, M.; Li, Y.; Lin, M.; Wang, N.; Wang, M.; Xiao, T.; Xu, B.; Zhang, C.; Zhang, Z. MXNet: A Flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv* **2015**, arXiv:1512.01274.
5. Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.* **2014**, *2014*, 2.
6. Rensin, D.K. *Kubernetes—Scheduling the Future at Cloud Scale*; O'Reilly Media: Sebastopol, CA, USA, 2015.
7. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Boston, MA, USA, 22–25 June 2010; USENIX Association: Berkeley, CA, USA; p. 10.
8. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113, [CrossRef]

9. Ousterhout, K.; Rasti, R.; Ratnasamy, S.; Shenker, S.; Chun, B.G. Making sense of performance in data analytics frameworks. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, USA, 4–6 May 2015; USENIX Association: Oakland, CA, USA; pp. 293–307.
10. Coates, A.; Huval, B.; Wang, T.; Wu, D.; Catanzaro, B.; Andrew, N. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*; Dasgupta, S., Mcallester, D., Eds.; JMLR Workshop and Conference: Atlanta, GA, USA, 2013; Volume 28, pp. 1337–1345.
11. Li, M.; Andersen, D.G.; Park, J.W.; Smola, A.J.; Ahmed, A.; Josifovski, V.; Long, J.; Shekita, E.J.; Su, B.Y. Scaling distributed machine learning with the parameter server. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, USA, 6–8 October 2014; USENIX Association: Broomfield, CO, USA; pp. 583–598.
12. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. In-datacenter performance analysis of a tensor processing unit. *arXiv* **2017**, arXiv:1704.04760.
13. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.D.; Katz, R.; Shenker, S.; Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, Boston, MA, USA, 30 March–1 April 2011; USENIX Association: Berkeley, CA, USA; pp. 295–308.
14. Douban Inc. TFMesos. Available online: https://github.com/douban/tfmesos (accessed on 7 March 2019).
15. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache hadoop YARN: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; ACM: New York, NY, USA; pp. 5:1–5:16.
16. Zaharia, M.; Borthakur, D.; Sen Sarma, J.; Elmeleegy, K.; Shenker, S.; Stoica, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*; ACM: New York, NY, USA, 2010; pp. 265–278. [CrossRef]
17. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.
18. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. *arXiv* **2015**, arXiv:1512.03385.
19. Krizhevsky, A. *Learning Multiple Layers of Features from Tiny Images*; Technical Report; University of Toronto: Toronto, ON, Canada, 2009.
20. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009.
21. Google. *Tensorflow Tutorial, Distributed Training*; Google: Mountain View, CA, USA, 2019.
22. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017.