Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems

Feng Yan
College of William and Mary
Williamsburg, VA, USA
fyan@cs.wm.edu

Olatunji Ruwase, Yuxiong He, Trishul Chilimbi
Microsoft Research
Redmond, WA, USA
{olruwase,yuxhe,trishulc}@microsoft.com

ABSTRACT

Big deep neural network (DNN) models trained on large amounts of data have recently achieved the best accuracy on hard tasks, such as image and speech recognition. Training these DNNs using a cluster of commodity machines is a promising approach since training is time consuming and compute-intensive. To enable training of extremely large DNNs, models are partitioned across machines. To expedite training on very large data sets, multiple model replicas are trained in parallel on different subsets of the training examples with a global parameter server maintaining shared weights across these replicas. The correct choice for model and data partitioning and overall system provisioning is highly dependent on the DNN and distributed system hardware characteristics. These decisions currently require significant domain expertise and time consuming empirical state space exploration.

This paper develops performance models that quantify the impact of these partitioning and provisioning decisions on overall distributed system performance and scalability. Also, we use these performance models to build a scalability optimizer that efficiently determines the optimal system configuration that minimizes DNN training time. We evaluate our performance models and scalability optimizer using a state-of-the-art distributed DNN training framework on two benchmark applications. The results show our performance models estimate DNN training time with high estimation accuracy and our scalability optimizer correctly chooses the best configurations, minimizing the training time of distributed DNNs.

1. INTRODUCTION

Deep neural network (DNN) models have recently attracted significant research and industrial interest because they achieve state-of-the-art accuracies on important artificial intelligence tasks, such as speech recognition [11, 15], image recognition [5, 8, 13, 21, 22], and text processing [7, 9, 10, 17]. An attractive feature of training DNNs (i.e., *deep learning*) is that their deep structure (number of layers) enables hierarchical feature learning, which is the key to achieving high accuracy but requires big DNN models trained on large quantities of data. In addition, task accuracy improves with increases in model size and amount of training data. This has enabled recent work [5, 13, 22] to achieve world-record accuracies on image recognition, but it requires training models with billions of connections using millions of images. Such large models do not fit on a single machine. Even if they do, training them would require several months. Moreover, high-accuracy models require good values for various neural network hyper-parameters and training parameters (e.g., learning rate, biases, etc.) that can only be determined empirically. Consequently, DNN training is an iterative process where the entire training procedure is repeated multiple times to tune DNN models to high accuracy. In addition, DNNs need to be retrained periodically to continuously incorporate new training data. Thus, faster DNN training is extremely important.

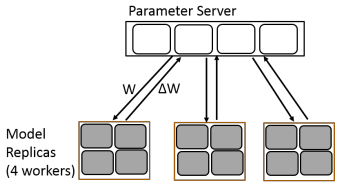

Figure 1: Architecture overview of distributed DNN.

To efficiently train large DNNs (billions of connections) to desired high accuracy using large amounts of data (terabytes) in a reasonable amount of time (several days), researchers have exploited distributed deep learning systems where the training is distributed over clusters of commodity machines [5, 13]. The DistBelief [13] and Adam [5] distributed deep learning systems run on commodity clusters of 1000 and 120 machines respectively connected by Ethernet. In addition to SIMD (single instruction multiple data) and thread parallelism on a single machine, these systems also exploit *model parallelism* and *data parallelism* across machines. Model parallelism partitions DNN across machines that we call *workers*. Each worker trains a portion of the DNN concurrently and a collection of workers that make up a DNN is called a *replica*. Data parallelism partitions training data to enable parallel training of multiple DNN replicas. To ensure convegence, replicas periodically exchange weight values through *parameter servers*, which maintains an updated global copy of the weights. Figure 1 shows an example of a distributed DNN system with 3 model replicas of 4 workers each, and with 4 parameter servers.

It is challenging to decide the appropriate configuration choices to efficiently train a large DNN using a distributed deep learning system. There are many different ways, e.g., thread parallelism, model parallelism, data parallelism, number of parameter servers, to partition and replicate the DNN across distributed hardware resources. Each combination of these parallelism knobs and their parallelism degrees represents a different system configuration choice, which can potentially produce dramatically different

KDD'15, August 10-13, 2015, Sydney, NSW, Australia.

DOI: http://dx.doi.org/10.1145/2783258.2783270.

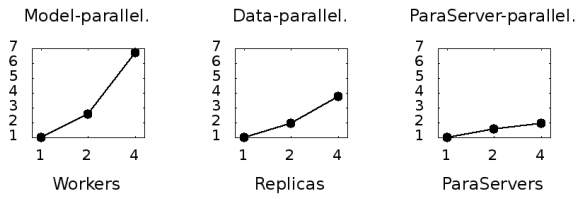

Figure 2: Scalability of DNN parallelism techniques. Each plot reports the speedup when scaling at only one dimension (fix the other two dimensions). See Section 6.1 for experimental setup.

scalability results. For example, Figure 2 shows how the parallelism techniques scale for training an image recognition DNN: model parallelism (the number of workers per model replica) is super-linear because of caching effects, data parallelism (the number of replicas) is roughly linear, and parameter server is diminishing as its communication bandwidth becomes less of a bottleneck. It is hard to estimate the performance impact of a system configuration as it depends on characteristics of the DNN (e.g., neuron count and connectivity) and hardware (e.g., machine count, clock speed, network latency and bandwidth). Moreover, these configuration choices lead to a multi-dimensional configuration space, which is very expensive to empirically explore for optimal solutions.

Configuring distributed hardware for efficient DNN training currently requires distributed system expertise which many machine learning researchers may lack, and even for system experts, the state space exploration is time consuming and they may settle for suboptimal solutions. Moreover, this expensive configuration process must be repeated if the application, DNN architecture, or hardware resources changes. To provide DNN training infrastructure as a service, an easier and more effective way of determining the performance impact of a system configuration for a specific DNN helps in selecting the best configuration that maximizes scalability and minimizes training time. *To address these issues, in this paper, we develop a performance model for estimating the scalability of distributed DNN training and use this model to power a scalability optimizer that determines the optimal distributed system configuration that minimizes training time.* We answer two key questions.

How much time does a system configuration take to train a DNN task? Our performance model takes as inputs the features of DNN and hardware, as well as the system configuration that maps the partitioned and replicated training work of the DNN to the available hardware. As outputs, it identifies scalability bottleneck and estimates the DNN training time. Our model supports the state-of-the-art design choices and quantifies the impact of different types of parallelism on computation and communication. We combine analytical modeling of the performance-critical components with a small set of guided system measurements capturing system behaviors (e.g., cache effects) that are hard to be modeled accurately.

What is the optimal system configuration to minimize DNN training time? Our scalability optimizer applies the performance models to explore various system configurations analytically, without requiring users to conduct exhaustive performance tests. We propose an efficient search algorithm that finds the optimal system configuration in polynomial time, significantly reducing the complexity from a brute-force search algorithm, whose computational cost grows exponentially with the number of DNN layers and machines.

We validate our approach by comparing our model's estimated training time and scalability for two DNN benchmarks, MNIST [23] and ImageNet [14] on a commodity cluster of 20 machines connected by 10Gbps Ethernet, with measurements from actual training runs on the cluster using the Adam distributed deep learning framework [5] . We show that our performance model estimates the training time of DNNs with high estimation accuracy. Among the tested configurations, we correctly identify the relative performance — a configuration that is considered faster by our model is also faster according to the measurements. The absolute training time estimated by our model is rather close to the measured value, with less than 25% difference. Moreover, our scalability optimizer correctly and efficiently finds the optimal configurations for both benchmarks. The experimental results demonstrate that the gap between different system configurations is large: an optimal configuration can be more than 20x faster than a reasonable configuration even when there are only 20 machines. This gap will only increase for larger-scale systems with more machines, highlighting the importance of our scalability optimizer. Finally, we use our model to predict how deep learning scales, both with more hardware, and custom hardware, such as FPGAs, ASICs, and RDMA.

This paper makes the following contributions:

- We develop a novel performance model for scalability estimation of distributed DNNs (Section 4).
- We build a scalability optimizer that efficiently searches and finds the optimal system configurations for distributed DNN training over a cluster of hardware resources, minimizing training time and maximizing system throughput (Section 5).
- We evaluate and validate the estimation accuracy of the performance model and the benefits of the scalability optimizer on a state-of-the-art deep learning framework with real-world benchmark applications (Section 6).

2. BACKGROUND

DNNs consist of large numbers of neurons with multiple inputs and a single output called an activation. Neurons are connected in a layer-wise manner with the activations of neurons in layer $l-1$ connected as inputs to neurons in layer l. The deep architecture of DNNs enables hierarchical features of the input data to be learned, making DNNs effective for difficult artificial intelligence (AI) tasks [3]. DNNs are typically trained using stochastic gradient descent (SGD) [4], where each input is processed in three steps: feed-forward evaluation, back-propagation and weight updates.

Feed-forward evaluation. Define a_i as the activation of neuron i in layer l. It is computed as a function of its J inputs from neurons in the preceding layer $l-1$:

$$a_i = f\left(\left(\sum_{j=1}^{J} w_{ij} \times a_j\right) + b_i\right) , \tag{1}$$

where w_{ij} is the weight associated with the connection between neurons i at layer l and neuron j at layer $l-1$, and b_i is a bias term associated with neuron i. The weights and bias terms constitute the parameters of the network that must be learned to accomplish the specified task. The activation function, f, associated with all neurons in the network is a pre-defined non-linear function, typically sigmoid or hyperbolic tangent.

Back-propagation. Error terms δ are computed for each neuron i in the output layer L:

$$\delta_i = (true_i - a_i) \times f'(a_i) , \tag{2}$$

where $true(x)$ is the true value of the output and $f'(x)$ is the derivative of $f(x)$. These error terms are back-propagated to each neuron i in the layer l from its S connected neurons in layer $l+1$:

$$\delta_i = \left(\sum_{s=1}^{S} \delta_s \times w_{si}\right) \times f'(a_i) . \tag{3}$$

Weight updates. These error terms are used to update the weights:

$$\Delta w_{ij} = \alpha \times \delta_i \times a_j \quad for\ j = 1...J\ , \tag{4}$$

where α is the learning rate and J is the number of neurons of the layer. This process is repeated for each input until the entire training data has been processed (i.e., a training *epoch*). Typically, training continues for multiple epochs, reprocessing the training data set each time, until the error converges to a desired (low) value.

Distributed DNNs are trained using asynchronous SGD, where shared weights are updated asynchronously and stale weights could be used for computation, to minimize communication costs. Prior work has shown learning accuracy is robust to asynchrony [5, 28], making it an important performance optimization. For example, replicas can run faster by synchronizing at a coarser granularity, after processing mini-batches of tens of inputs [6, 16].

3. MODELING OBJECTIVE AND USE CASES

In practice, large DNN models are often trained on large amounts of data through "trial and error" tuning of network hyper-parameters and training parameters (e.g., biases, learning rate, etc.), a process which can benefit from distributed hardware for accelerating each trial. Distributed training, however, poses new challenges and requires tuning an additional set of parameters — distributed system configurations. Since many machine learning researchers may not be experts at configuring distributed hardware, our work aims to help them pick the best configurations of available hardware resources to expedite distributed DNN training. And even for system experts, our work helps avoid the time consuming state space exploration for optimal system configurations for each DNN and hardware combination.

Our performance model estimates training epoch time for a given system configuration, and identifies configurations which minimize epoch time. We do not model the convergence to desired accuracy. To the best of our knowledge, it is notoriously difficult to model and bound accuracy for the hard non-convex learning problems [12] (e.g., image and speech recognition), which benefit the most from large-scale distributed deep learning and thus are the focus of our work. We show our performance models help speed up the standard "trial and error" process of training these hard DNN tasks.

The process of finding good values of hyper-parameters (and training parameters) typically proceeds as follows. You pick an initial setting for the hyper-parameters, and then use a distributed framework to run the training procedure for a fixed amount of time (e.g., days) on some configurations of the hardware resources. If the final accuracy is not satisfactory, then choose another set of parameter values and repeat the process, until desired accuracy is achieved. Given the iterative nature of this process, system configurations with faster epochs are preferable as they can complete more epochs within the time budget, and accuracy improves with more epochs. This is important when training large DNN models on large amounts of data, where convergence may take too long (e.g., months) and the goal of each trial is often to achieve maximal accuracy within a time budget (e.g., a week).

For a given set of hyper-parameters, different system configurations can also result in different efficiency and accuracy values. Figure 3 shows the training efficiency and accuracy of 66 different sets of system configurations of using 20 machines for ImageNet-100, the 100 categories image classification task of the ImageNet benchmark [14]. The hardware and benchmark details are provided in Section 6.1. The x-axis depicts the epoch time, where the values are normalized against the epoch time of the fastest configuration. The Y-axis depicts the accuracy after each configuration

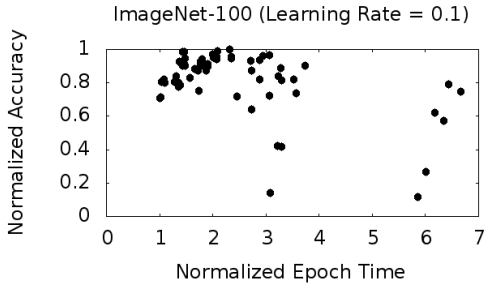

Figure 3: Accuracy and time of different system configurations.

runs for 10 epochs[1], where the values are normalized against the highest accuracy among all configurations. Each point in the figure represents the efficiency and accuracy results of one system configuration. Figure 3 shows that, the epoch time of the fastest and the slowest configuration differs by 6.7X; while the accuracy of the configurations differs by 8.7X. Very importantly, among different configurations, it is hard to pinpoint any correlation between the efficiency and accuracy of a system. In other words, there are systems both efficient and accurate, both inefficient and inaccurate, either efficient or accurate. The accuracy of a system configuration can only be evaluated empirically.

To find a system configuration achieving high accuracy using less time, a sensible approach is to try out the more efficient systems, if they provide accurate results, the system is both efficient and accurate, and even if some of them do not achieve high accuracy, we spent less time to find the right set of configuration choices than starting from a random or inefficient system. To illustrate this, we present a case study using the example in Figure 3. Assume the desired accuracy is 0.9. The typical trial and error method randomly picks up a system configuration to test whether it achieves the desired accuracy. Since the time to find a proper configuration of such method depends on how the configurations are selected, we present here the expected time by average over 10 different training runs of using uniformly-random selection. The expected normalized time to achieve the desired accuracy is 353.56. In comparison, searching from the configuration with the smallest epoch time only takes 129.80 normalized time to find a configuration that meets the desired accuracy, which is about 1/3 of the time by using random selection. These usage cases motivate our study: build performance models to estimate system efficiency for given configuration choices (Section 4), and develop optimization procedures to find the most efficient configurations (Section 5).

4. PERFORMANCE MODEL

This section presents the performance models of distributed DNNs. We cover the state-of-the-art design and configuration choices supported by distributed DNN infrastructures [5, 13]. We quantify the impact of these choices on the DNN training time. Section 4.1 focuses on model parallelism, where we estimate the training time of a single input with partitioned neural networks across multiple machines. Section 4.2 models three forms of data parallelism, where multiple inputs are trained concurrently. We integrate model and data parallelism into a complete performance model for estimating the epoch time of DNNs with any given system configurations. The appendix of the tech-report [31] summarizes the key notations used in the performance model and the rest of the paper.

[1]ImageNet-100 converges after 10 epochs, so the accuracy reported here is the final accuracy.

4.1 Model Parallelism

Model parallelism partitions a DNN layer across multiple machines[2] so that each partition can be trained independently, with activations of neural connections that cross machine boundaries being exchanged as network messages. The number of partitions is a configuration choice: a larger value increases aggregate cache capacity and bandwidth, but incurs additional communication from cross-machine neural connections. This section quantifies the impact of model parallelism: we estimate the training time of a single sample under different numbers of neural network partitions. Here we call each piece of inputs as a *sample*, e.g., at an image recognition task, a sample is an input image and its label.

As DNN consists of different types of layers with varying connectivity, an appropriate number of partitions could vary across different layers of one DNN task. For example, DNNs for image processing often comprises *convolutional* layers (possibly interleaved with *pooling* layers) at the bottom followed by *fully connected* layers. Convolutional layers are only connected to spatially local neurons in the lower layer, modeling the visual cortex [18]; pooling layers summarize the salient features learned by convolutional layers; whereas neurons in fully connected layers are connected to all neurons in the previous layer. Therefore, it could be beneficial to partition convolutional and pooling layers (more aggressively) as the number of cross-machine connections is smaller, while partitioning a fully connected layer could generate more communication cross machines, which may not speed up the training time. Although prior work applies the same number of partitions for an entire neural network [5], our work supports a more general model that allows different number of partitions at different layers to fully exploit model parallelism.

Figure 4 visualizes DNN partitions for model parallelism. The DNN has L layers. Each layer $l \in [1, ..., L]$ is partitioned into $P(l)$ segments, and each segment is denoted by p and $p \in [1...P(l)]$[3]. The segments of a layer are processed in parallel, thus the layer execution time is decided by the slowest segment. To reduce training time and improve system utilization, the segments of the same layer are often evenly partitioned, i.e., each segment has roughly the same number of neurons and connections to balance the computation and communication among the identically configured machines (our model considers homogeneous hardware). Therefore, the time spent on a layer l can be estimated using any of its segment p. We calculate the training time of DNN of a sample as the summation of its time spent in each layer. The total time in each layer is composed of the time spent on feed-forward evaluation, back-propagation, and weight updates, each of which is further divided into computation and communication time.

4.1.1 Feed-forward Evaluation

For each segment p at layer l, the feed-forward evaluation time $T_f(l,p)$ is equal to the time spent in computing the output activations of the neurons in the segment (denoted as $U_f(l,p)$) and communicating activations from the connected segments in layer $l-1$ (denoted as $M_f(l,p)$), i.e., $T_f(l,p) = U_f(l,p) + M_f(l,p)$.

Computation. Feed-forward evaluation computes the output activations of the neurons in each layer. As shown in Eq. 1, the output activation of neuron i in layer l is the result of a nonlinear function $f(x)$, where the input x is the dot product of the input activations of

[2]Our performance model can also handle model parallelism using chip-level multiprocessing, but is not discussed due to space limits.

[3]Layers can be partitioned in different ways (e.g., stripes, fixed-size squares, etc.) which impacts the computation and communication load per partition. Our approach applies to any partitioning scheme, but for clarity we assume stripe partitioning in the rest of this paper.

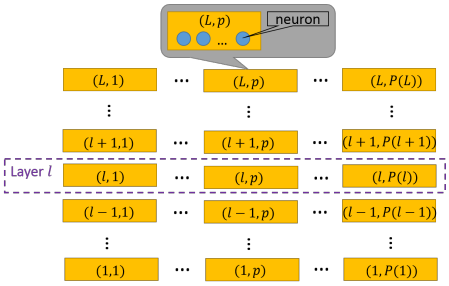

Figure 4: DNN partitions for model parallelism.

i from layer $l-1$ and the weight values of the connections. Thus, the computation time per neuron is equal to $C_{muladd} \times |\mathcal{S}_i| + C_{act}$, where $\mathcal{S}_i$ represents the set of neurons at layer $l-1$ connected to neuron i of layer l, C_{muladd} denotes the time of one multiply-add operation, and C_{act} denotes the time to compute $f(x)$. We estimate the computation time of segment p as:

$$U_f(l,p) = C_{muladd} \times W(l,p) + C_{act} \times N_{neuron}(l,p)\ , \quad (5)$$

where $N_{neuron}(l,p)$ denote the number of neurons in segment p, and $W(l,p)$ denote the number of weights connected from layer $l-1$ to all neurons in segment p.

A first-order estimation of C_{muladd} and C_{act} that captures the cache effects of model parallelism can be obtained through profiling a micro benchmark that emulates feed-forward evaluation using these basic operations (i.e., canonical computation) on a worker machine. Figure 5 shows an example of a simple canonical feed-forward evaluation. Estimation accuracy of the canonical code can be improved by incorporating optimizations from real-world DNN implementations (e.g., loop tiling and SIMD arithmetic).

```
for (i = 0; i < N_neuron(l,p); i++) {
  foreach (j ∈ S_i){
    y_i += w_ij * a_j   // cost: C_muladd
  }
  a_i = f(y_i)   // cost: C_act
}
```

Figure 5: Canonical feed-forward evaluation.

Communication. Since activations can be sent asynchronously, we assume the communication time of feed-forward evaluation is dominated by the delay in receiving cross-machine activations from previous layer. Thus, the communication time $M_f(l,p)$ is a function of the data size received by p and the network performance:

$$M_f(l,p) = C_{ncost} + \frac{A(l,p) \times C_{bits}}{C_{nbw}}\ , \quad (6)$$

where C_{ncost} is network latency of sending one bit of data between two workers, C_{nbw} is the bandwidth of the machine's NIC, $A(l,p)$ is the number of remote activations that the segment p receives from layer $l-1$, and C_{bits} is the size of each activation. $A(l,p) \times C_{bits}$ is the number of data bits received by p.

4.1.2 Back-propagation

Computation. Back-propagation computes the error terms of neurons. The error term of neuron i in segment p of layer l is computed from the input error terms of i in layer $l+1$, the connection

weights, and the error function $f^{'}(x)$, as shown in Eq. 3. We estimate the computation time of segment p as:

$$U_b(l,p) = C_{muladd} \times W^{'}(l,p) + N_{neuron}(l,p) \times C_{err} \ , \quad (7)$$

where C_{err} is the basic cost of the error function, and $W^{'}(l,p)$ is the number of connections from layer $l+1$ to segment p of layer l. We estimate the cost of the basic operations through canonical computations, similar to Figure 5.

Communication. We assume the back-propagation time of segment p in layer l, $M_b(l,p)$, to be the delay in receiving remote error terms from layer $l+1$. Thus, $M_b(l,p)$ can be estimated using a similar equation to Eq. 6, but with $A(l,p)$ replaced by $E(l,p)$, the number of remote error terms:

$$M_b(l,p) = C_{ncost} + \frac{E(l,p) \times C_{bits}}{C_{nbw}} \ .$$

4.1.3 Weight Updates

Error terms are propagated through the neural network to update the weight values in each layer. As shown in Eq. 4, the delta weight Δw_{ij} for the connection between neuron i in layer $l+1$ and neuron j in layer l is computed from the error term δ_i and the activation a_j. The weight value w_{ij} is then updated using Δw_{ij}. Thus, the computation time for weight updates is estimated as

$$U_w(l,p) = C_{muladd} \times W(l,p) \ . \quad (8)$$

Note that weights are not communicated in model parallelism, thus communication time, $M_w(l,p)$, is zero.

From the estimated time for feed-forward evaluation, back-propagation, and weight updates, we obtain the training time for each layer, and thus the total time to train on an example with model parallelism. Our models are applicable to layers of different types, such as pooling, convolutional, and fully connected layers[4], by estimating computation time using appropriate canonical forms, and communication time using the size of remote activation/error terms.

4.2 Data Parallelism

Rather than processing training samples one by one, data parallelism accelerates training through concurrent processing of multiple samples. We model 3 forms of data parallelism through CMP (chip-level multiprocessing), by layer replication, and using model replicas with parameter servers. This section presents a complete performance model that builds these 3 forms of data parallelism on top of the model parallelism.

4.2.1 Chip-level Multiprocessing

Exploiting the modern multi-core processors, the cores of a CMP system process different samples concurrently(e.g., a 16 core processing 16 samples at a time), while asynchronously sharing weights through shared memory. The number of concurrent threads is a configuration choice: a higher value increases concurrency but also increases the potential interference among the threads. Figure 6 extends our model to support data parallelism using CMP. We add one more dimension h ($h \in [1, H(l)]$) to the base model, where $H(l)$ represents the number of threads training in parallel in layer l. This extends our index of each segment from (layer ID, partition ID) pair to a triple of (layer ID, partition ID, thread ID).

Computation. Concurrent training of multiple samples may interfere with each other, competing for memory bandwidth, and thus affecting per-sample computation time. We define a performance interference factor $C_{interf}(H(l))$ to model the interference among

[4]Recurrent neural networks [25] can be viewed as a DNN unfolding over time, so our performance model is easily applicable to it.

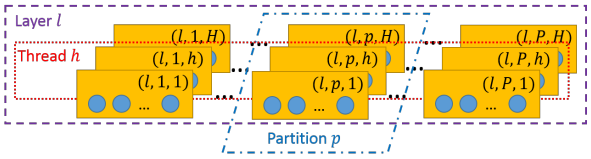

Figure 6: Data Parallelism using CMP.

$H(l)$ threads. We estimate $C_{interf}(H(l))$ by running a multi-threaded version of the canonical form such that each thread processes the same code segment using different cores: $C_{interf}(H(l))$ is estimated as the ratio of the $H(l)$-thread execution time and the single-thread execution time. Thus, the computation time of a segment using CMP of $H(l)$ threads is:

$$U_{i=\{F,B,W\}}(l,p,h) = C_{interf}(H(l)) \times U_i(l,p) \ , \quad (9)$$

where $U_i(l,p)$ is the computation time of having one thread per layer, as shown in Eq. 5, Eq. 7 and Eq. 8.

Eq. 9 shows data parallelism may not reduce the computation time of an example, on the contrary, it may increase the time due to the potential interference among threads. However, running multiple samples concurrently can still reduce the epoch time T_{epoch} of training the entire sample set once (total of N_{sample} samples). Thus, we define *data parallelism degree* $Q(l)$ as the concurrency degree of training multiple samples in parallel at layer l. With $H(l)$ concurrent threads that each runs a sample, we have $Q(l) = H(l)$ at layer l. Per-sample execution time and data parallelism degree together define the epoch time and system throughput.

Communication. When training multiple samples with multiple threads, the network bandwidth is shared among threads, i.e., each thread gets $C_{nbw}(H(l)) = 1/H(l)$ of the bandwidth. We model the network latency $C_{ncost}(H(l))$ as a function of $H(l)$ as this latency may increase when both sender and receivers establish $H(l)$ concurrent connections. The communication time is estimated as:

$$M_f(l,p,h) = C_{ncost}(H(l)) + \frac{A(l,p,h) \times C_{bits}}{C_{nbw}(H(l))} \ ,$$

$$M_b(l,p,h) = C_{ncost}(H(l)) + \frac{E(l,p,h) \times C_{bits}}{C_{nbw}(H(l))} \ .$$

4.2.2 Layer Replication

Replicating a layer across the worker machines of a model replica can reduce the network overheads of cross-machine activations. Since a fully connected layer can be expensive to partition, a faster alternative is to replicate it among several machines where each machine processes a subset of training data. We add a new dimension r to represent replication. This extends our index of each segment to a quadruplet of (layer ID, partition ID, thread ID, replication ID). The computation time is the same as that in Section 4.2.1, but the data parallelism degree is extended to $Q(l) = H(l) \times R(l)$, where $R(l)$ denotes the number of replications for layer l. The communication is calculated similarly as that in Section 4.2.1. We show an example of output layer replication in Section 6.4.

4.2.3 Multiple Model Replicas

The third form of data parallelism is partitioning the training data to train the model replicas in parallel, where the replicas share weights through a parameter server (Figure 1). To improve parameter server throughput, the weights are partitioned across multiple servers for more network bandwidth. The number of model replicas and parameter server machines are critical configuration choices to balance computation and communication, weight communication

load scales with model replica count. But, an optimal ratio between them is task specific, depending on both the DNN and hardware.

Computation. The computation time per sample does not change. The data parallelism degree is extended to $Q(l) = H(l) \times R(l) \times N_{mr}$, where N_{mr} denotes the number of model replicas.

Communication. The difference in communication comes from introducing the parameter servers. Writing weights to parameter servers can be done asynchronously and does not affect training time on the replicas. However, reading weights is synchronous and can stall training until the read is complete. Therefore, the reading time should be considered in the model. We use N_{read} to represent the read frequency, i.e., each replica reads weights from parameter servers after training N_{read} samples.

The communication time between replicas and parameter servers varies depending on their communication patterns of the actual implementation. The worst case is when all replicas read weights from the same parameter server simultaneously: the bandwidth of the parameter server to each replica becomes minimum. The time to read weights from parameter servers is:

$$M_w^{max} = \sum_{l=1}^{L} (C_{ncost}(H(l)) + \frac{N_{mr} \times W(l)}{C_{nbw}(H(l))}) , \qquad (10)$$

where $W(l) = \sum_{1 \le p \le P(l)} W(l,p)$ is the data size of all weights for layer l. The best case occurs when (1) a single replica uses all its N_{wr} workers and reads from all parameter servers in parallel using their accumulated bandwidth; and (2) there is no overlapped time among multiple replicas while reading weights.

$$M_w^{min} = \sum_{l=1}^{L} (C_{ncost}(H(l)) + \frac{W(l)}{C_{nbw}(H(l)) \times \min(N_{ps}, N_{wr})}) . \qquad (11)$$

In practice, the weight reading time M_w is somewhere in between depending on the implementation. There are optimization opportunities to improve the reading time to be closer to the best case. For example, the reading time for different replicas can be scheduled so that there is minimal overlap.

We estimate the epoch time of the complete model by considering the worst case where the computation of model replicas does not overlap with their communication to parameter servers:

$$T_{epoch} = \frac{M_w \times N_{sample}}{N_{read}} + \sum_{l=1}^{L} \{ [\, U_f(l,p,h,r) + M_f(l,p,h,r) + U_b(l,p,h,r) + M_b(l,p,h,r) + U_w(l,p,h,r) \,] \times N_{sample}/Q(l) \} .$$

5. SCALABILITY OPTIMIZATION

In this section, we develop a scalability optimizer that enumerates different system configurations, uses the proposed performance model to estimate their training time, and finds an optimal system configuration with minimum epoch training time. Compared with a brute-force search algorithm whose complexity is exponential, our algorithm finds optimal configuration efficiently in polynomial time, making the solution computationally trackable in practice.

5.1 Problem Formulation

We define the constrained optimization problem: for a DNN, its training data, and a given cluster of N machines, find an optimal system configuration such that the epoch training time is minimized. Figure 7 presents the formal problem definition.

Defining a system configuration requires three sets of parameters: (1) the parameters defining resources, including the number of parameter servers N_{ps}, model replicas N_{mr}, and workers N_{wr} per model replica; (2) the parameters defining the number of partitions, threads and replications at each layer of DNN[5]; and (3) the mapping function f between the resources and the segments of DNN, in particular, between the workers of each model replica and its partitioned and replicated segments.

Variables to define a configuration Φ
- N_{ps}: number of parameter servers
- N_{mr}: number of model replicas
- N_{wr}: number of workers per model replica; $\mathcal{SW}$=$\{1, ..., N_{wr}\}$

For each layer $1 \le l \le L$,
- P_l: number of partitions; $\mathcal{SP}_l$=$\{1, ..., P_l\}$
- H_l: number of threads; $\mathcal{SH}_l$=$\{1, ..., H_l\}$
- R_l: number of replicas; $\mathcal{SR}_l$=$\{1, ..., R_l\}$
- mapping function f_l: $\mathcal{SP}_l \times \mathcal{SH}_l \times \mathcal{SR}_l \rightarrow \mathcal{SW}$

Objective:

minimize$_{\{\Phi\}} T_{epoch}(\Phi)$

Subject to:

C1: $N_{ps} + N_{mr} \times N_{wr} \le N$

C2: $H_l \le K$, for all $1 \le l \le L$ (K is total number of cores of a machine)

C3: $f_l(p,h,r) \ne f_l(p',h',r')$, if $p \ne p'$ or $r \ne r'$, for all $1 \le l \le L, p, p' \in \mathcal{SP}_l, r, r' \in \mathcal{SR}_l$ and $h, h' \in \mathcal{SH}_l$

C4: $f_l(p,h,r) = f_l(p,h',r)$, for all $1 \le l \le L$, $p \in \mathcal{SP}_l, r \in \mathcal{SR}_l$ and $h, h' \in \mathcal{SH}_l$

Calculate $T_{epoch}(\Phi)$ for any given Φ using models in Section 4.1 and 4.2

Figure 7: Problem Formulation.

There are 4 sets of constraints. C1 bounds the total number of machines used by the DNN task. We assume all machines are the same in terms of computing and communication performance. C2 bounds the total number of concurrent threads per machine by the total number of cores of the machine to avoid cache contention and context switch among threads. C3 ensures that, at each layer, two segments of different partitions or replications are NOT mapped to the same worker. C4 ensures that the segments exploiting chip-level multiprocessing (with the same partition and replication IDs) are mapped to the same worker. Given a system configuration Φ, we can then calculate the training epoch time of the DNN using the performance model proposed in Section 4. The optimization objective is to find the configuration with minimum epoch time.

5.2 Challenges

To solve this optimization problem, a simple way is to do a brute-force search on all combinations of configurations. However, the search space is very large as shown below.

Considering different selection of the resource parameters (N_{ps}, N_{mr}, N_{wr}), there are at most N^3 combinations. For each combination, inside a model replica, we explore system configurations along two directions:

Segment-worker mapping. There are different ways to assign segments to workers in each layer. This mapping becomes more complex as different layers may require a different number of segments and workers. To find the assignment that minimizes the remote communications is a permutation of segments and workers, which results in $O(N!)$ combinations.

Multi-layer composition. DNN often consists of multiple layers and each layer can make different decisions. At each layer, there are up to N^2 combinations on the selection of the number of partitions and the number of replications, and K choices on the number

[5]We assume the number of threads, replications, and partitions are the same for all layers to balance layer training time among workers: the slowest worker determines layer time. However, the number of partitions, replications and threads can differ across layers.

of threads per worker. Adding the possible mapping of workers to segments, there are $K \times N^2 \times (N!)$ combinations for each layer. For a DNN with L layers, the entire search cost is

$$O(N^3[\ K \times N^2 \times (N!)\ ]^L)\ ,$$

which is exponential with respect to the number of machines and the number of layers. This brute force search is too computationally expensive to be useful in practice.

5.3 Efficient Search Algorithm

Our optimal polynomial-time search algorithm is based on two insights: (1) a greedy approach that decides the best worker for each segment, reducing the cost of segment-working mapping, and (2) constructing optimal solution using optimal subproblems via dynamic programming, reducing multi-layer composition costs.

Optimizing segment-worker mapping. Finding an optimal mapping of all segments to workers per layer by exhaustively trying all combinations is an expensive permutation problem. However, our key observation is that, a greedy approach that decides the best worker for each segment represents the optimal global mapping. More precisely, for each segment, we find a worker that minimizes the remote connection of the segment from previous layer, which can be done efficiently with time complexity of $O(N)$. Therefore, to compute the best workers for all the segments, the complexity is $O(N^2)$. This optimization reduces the search complexity to $O(N^3[\ K \times N^4\ ]^L)$. Because of the symmetric and flat structure of DNNs, the case where two segments have the same best worker is very unlikely to happen. And when it happens, we can break the tie easily by comparing which mapping choice results in overall smaller number of remote connections. Even in the worst case that we assume the special property of DNN does not help reduce the likelihood of ties, we can map the problem to a general maximum weighted bipartite matching problem, where the optimal solution can be computed in polynomial time with complexity of $O(N^4)$ [30]. Due to interest of space, we defer the detailed analysis and example to our tech-report [31].

Optimizing multi-layer composition. We apply dynamic programming to find optimal solution while reducing the computational cost. Our key observation is that an optimal solution for up to l layers can be constructed using the optimal of sub-problems for up to layer $l-1$ [6]. To ease the presentation, we first consider the case with partition only. Let's denote $T_{epoch}([1,l],p_l)$ as the optimal accumulated training time from layer 1 to layer l where the layer l has p_l partitions. If we know the optimal solution $T_{epoch}([1,l-1],p_{l-1})$ of the subproblems at layer $l-1$, we can compute the optimal of up to layer l by selecting among the configurations at layer $l-1$ that minimizes the sum of the computation time at layer l, the communication time between layer $l-1$ and l, and the accumulated training time up to $l-1$, as follows:

$$T_{epoch}([1,l],p_l) = \min_{1\le p_{l-1}\le N}[T_{epoch}([1,l-1],p_{l-1}) + U(l,p_l) + M(l,l-1,p_l,p_{l-1})]\ ,$$

where $U(l,p_l)$ is the computation time of layer l with p_l partitions, and $M(l,l-1,p_l,p_{l-1})$ is the communication time between layer $l-1$ and l when there are p_l partitions in layer l and p_{l-1} partitions in layer $l-1$.

[6]Note that a greedy approach that minimizes the cost of each layer is suboptimal: the number of partitions at layer l is affected by that at layer $l-1$ and affects layer $l+1$. Minimizing the cost of one layer greedily may increase the costs of other layers, rendering suboptimal decisions for the entire network.

The base case is the first layer, which has no communication from previous layer, i.e., $T_{epoch}([1,1],p_1) = U(1,p_1)$. The optimal epoch time of the entire system is computed as

$$T^*_{epoch} = \min_{1\le p_l\le N} T_{epoch}([1,L],p_l)\ .$$

If we solve this problem recursively, the cost is exponential. We apply dynamic programming technique to reduce the computation cost. More specifically, we compute and store the optimal solution for subproblems, starting from layer 1, and going upward to layer L. At each layer l, we compute $T_{epoch}([1,l],p_l)$ from $p_l = 1$ up to $p_l = N$. To get each $T_{epoch}([1,l],p_l)$, the cost is N times the corresponding segment-worker mapping cost, i.e., N^3. If we also consider replication, the cost per layer becomes $O(N^4)$ and the total cost of L layers is $O(L \times N^4)$. To sum up, the entire cost after applying these two techniques is:

$$O(L \times K \times N^7)\ .$$

This complexity can be further reduced to $O(L \times K \times N^5 \log^2 N)$.

Combining the two techniques, we effectively reduce the complexity of computing optimal solution from exponential time to polynomial time, making it computationally feasible in practice.

6. EVALUATION

We now evaluate our proposed performance models and scalability optimizer for distributed DNN training on a commodity computing cluster. Our experiments validate the estimation accuracy of the models and illustrate the benefits of the optimizer. We also use our models to predict DNN training time with larger commodity clusters and custom hardware, such as FPGAs and ASICs.

6.1 Methodology

We now describe the distributed deep learning framework, benchmark applications, and computing cluster used in our experiments.

Distributed deep learning system. Adam is a state-of-the-art framework for distributed training of large DNNs (billions of connections) using vast amounts of training examples on a cluster of commodity machines [5]. We use measured results from Adam to validate the estimation accuracy of our performance models.

Benchmarks. We use two popular benchmarks, *MNIST* [23] and *ImageNet* [14], with corresponding DNNs [5] that achieve state-of-the-art task accuracy. The MNIST task is to classify 28x28 images of handwritten digits into 10 categories using 60000 training examples [23]. The MNIST DNN is relatively small, containing about 2.5 million connections in 5 layers: 2 convolutional layers, 2 linear layers, and a 10-way softmax output layer. The ImageNet task is to classify 256x256 images from a dataset of over 15 million images into 22,000 categories (a.k.a., *ImageNet-22K*). The *ImageNet-22K* DNN is extremely large, containing over 2 billion connections in 8 layers: 5 convolutional layers, 2 linear layers, and a 22000-way softmax output layer. These DNNs represent 2 extremes of the size spectrum and provide insights on the generality of our models.

Computing Cluster. We use a cluster of 20 identically configured commodity servers, connected by Ethernet. The servers run a 64-bit Windows Server 2012 DataCenter OS, with the benchmark datasets preloaded on local disk. Each server is a dual-socket Intel Xeon E5-2450 system with 16 cores running at 2.1GHz, 64 GB of memory, 268.8 GFLOP/s SIMD FPU, and a single 10Gbps (bidirectional) NIC. 16 training threads are enabled per server.

6.2 Performance Model Validation

We validated our performance models by measuring the estimation accuracy of the scalability (training speedup) of model paral-

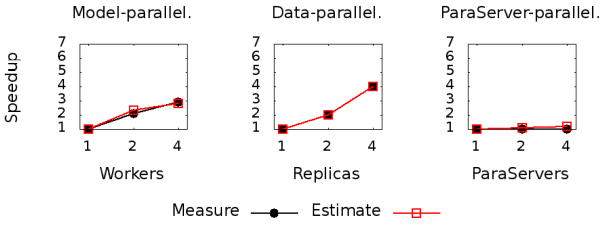

Figure 8: Scalability results of MNIST.

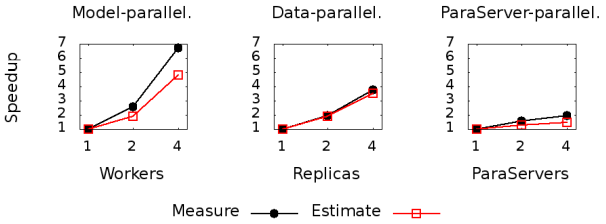

Figure 9: Scalability results of ImageNet-22K.

lelism[7], data parallelism using replicas, and parameter server parallelism, and the relative performance of system configurations arising from different combinations of these parallelism modes. Unless specified otherwise (Section 6.3), a model replica writes and reads the parameter servers after processing 40 and 70 images respectively. We compare estimations from our performance model (labelled *Estimate*) to measurements from Adam (labelled *Measure*).

6.2.1 *Estimating Scalability*

We configure distributed training scenarios that correspond to 1, 2, and 4 parallelism degrees for one mode and a parallelism setting of 4 for the others. The training speedup results are summarized in Figures 8 and 9. The results show that the 3 parallelism techniques scale differently for a given workload, and a given technique can scale differently across workloads. For MNIST, model parallelism speedup is linear from 1 to 2 workers and sub-linear from 2 to 4 workers, data parallelism speedup is always linear, while parameter server parallelism provides no speedup. For ImageNet-22K, model parallelism speedup is super-linear, while data and parameter server parallelism speedups are roughly linear.

These results lead to 3 observations. (1) Model parallelism benefits ImageNet-22K more than MNIST because partitioning makes the ImageNet-22K model fit into L3 cache, while the smaller MNIST model fits in L3 without partitioning. (2) Data parallelism scales linearly as long as the parameter server is not a bottleneck. (3) The benefits of parameter server scaling depends on the amount of bottleneck caused by replica synchronization, which is a function of DNN size. For MNIST with 4 workers and 4 replicas, 1 parameter server is sufficient, and thus parameter server scaling gives little speedup to replica synchronization. For ImageNet-22K, the extra network bandwidth of a second parameter server linearly speeds up replica synchronization, but more servers give diminishing speedups. The results show that our performance model accurately estimates the different scalability behaviors of training tasks.

6.2.2 *Estimating Relative Training Speed*

We measured the estimation accuracy of our performance models in the relative training speed for different system configurations,

[7]The DNN architectures restricted us to model-parallelism of ≤ 4.

MNIST		
Configuration	Rank	
#W/#R/#PS	Measure	Estimate
2/9/2	1	1
2/8/4	2	2
2/6/8	3	3
4/4/4	4	4
1/19/1	5	5
1/18/2	6	6
1/16/4	7	7
1/12/8	8	8

(a)

ImageNet-22K		
Configuration	Rank	
#W/#R/#PS	Measure	Estimate
4/4/4	1	1
4/4/2	2	2
2/9/2	3	3
4/3/6	4	4
4/3/4	5	5
2/8/4	6	6
2/6/6	7	7
2/6/4	8	8

(b)

Table 1: Training speed ranking (1 is the fastest). W, R, and PS stand for worker, replica, and parameter server respectively.

which is useful for head-to-head comparison of competing configurations. For each benchmark we studied 8 configurations which use as much of the available 20 machines as possible, and are interesting from a scalability perspective based on Section 6.2.1.

Tables 1 presents results of the 8 configurations for MNIST and ImageNet-22K. We see that our performance model provides highly accurate estimations: the relative rankings of the 8 configurations from our models exactly match the measured results for both benchmarks. Moreover, the best configurations for the benchmarks are different: the best for MNIST is <2 workers, 9 replicas, 2 parameter servers>, and for ImageNet-22K is <4 workers, 4 replicas, 4 parameter servers>. Our models successfully quantify the difference in the computation and communication patterns of these workloads, and thus identify the respective best configurations.

A closer examination of Table 1 brings interesting observations: why is <2 workers, 9 replicas, 2 parameter servers> faster than <1 workers, 18 replicas, 2 parameter servers>? Intuitively, given linear scaling of model parallelism (from 1 to 2) and linear scaling of data parallelism in Figure 8, should these two configurations have the same performance? It turns out that their communication times with parameter servers differ. In particular, a single-worker replicas does not effectively leverage the extra parameter server bandwidth (as shown in Eq. 11, the available bandwidth is bounded by the smaller of the parameter server bandwidth and model replica bandwidth). This is also reflected by the configurations with 12 or more replicas of 1 worker: performance degrades by increasing the number of parameter servers at the expense of replicas. Such factors might be neglected if we simply estimate training time with a sketch, however, our models capture them nicely, showing the benefits of a comprehensive performance model.

6.3 Training Time Estimation Zoom In

We now use ImageNet-22K for a more detailed evaluation of the estimation accuracy of our performance model in terms of absolute training speed and the effect of varying the rate of replica synchronization. MNIST results are similar and skipped for space.

Absolute Training Time and Breakdown. The absolute epoch training times of ImageNet-22K for different system configurations are presented in Figure 10 . We can see that our performance model estimates training time fairly accurately ($< 25\%$ estimation error). Figure 11 presents a detailed breakdown of the estimation errors for each computation and communication component in the feed-forward evaluation, back propagation, and weight update steps. We observe that we estimate computation rather accurately, but the communication time, due to complex system behaviors on network, is hard to model accurately. For example, the communication time M_w between a model replica and parameter servers is bounded by M_w^{max} (Eq. 10) and M_w^{min} ((Eq. 11). However, the actual time depends on weights distribution across the parameter servers, the scheduling of model replicas to read weights, and probability of collisions, which are hard to estimate exactly. Nevertheless, our

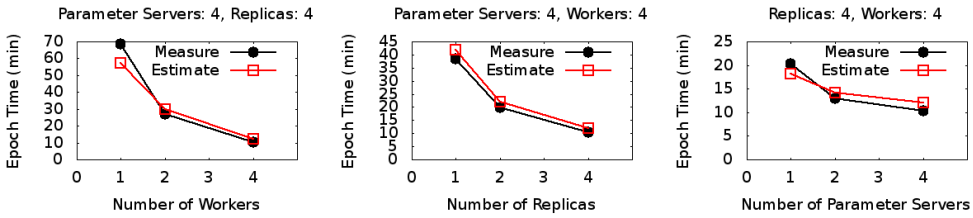

Figure 10: Estimated and measured epoch training time in different scenarios of ImageNet-22K.

models accurately estimate the relative speed of different configurations, which is more crucial to our design goals.

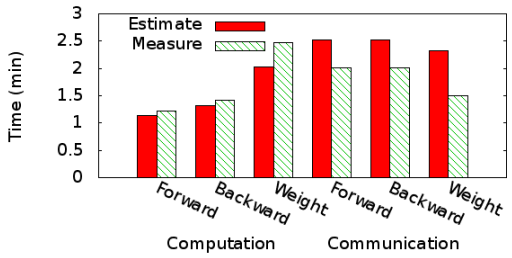

Figure 11: Epoch training time breakdown of ImageNet-22K.

Effect of Replica Synchronization. The impact of varying the rate of reading and writing parameters from the parameter server is illustrated by Figure 12. The results are shown in terms of time to complete a training epoch. As expected, faster training and better scalability is achieved with less frequent synchronization (e.g., <80, 150> read/write ratio yields the shortest time). Our performance model captures the correlation between synchronization rates and training time, and estimates the scalability accurately.

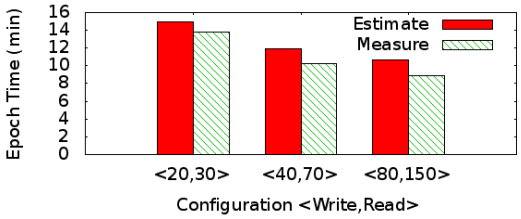

Figure 12: Effect of replica synchronization rate on ImageNet-22K.

While performance improves with less synchronization (less time per epoch), the learning rate (i.e., accuracy improvement per epoch) may decrease (more epochs to achieve a target accuracy) [5]. Exploiting the tradeoff between the performance and accuracy with varying degree of asynchrony could be an interesting future work.

6.4 Optimization

We illustrate the benefits of the optimizer with two experiments.

Partitioned vs Replicated Fully Connected Layer. To show how our performance models can guide micro-level optimizations of distributed training, we consider the design choice of partitioning or replicating fully connected layers. Both designs are illustrated in Figure 13 with a 2-worker replica for training a 3-layer DNN with partitioned and replicated fully-connected output layer. With partitioning, each worker performs half the computations and activation communications of the ouput layer for each image. With

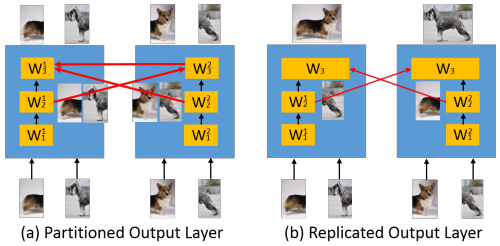

Figure 13: Example of partitioned and replicated output layer.

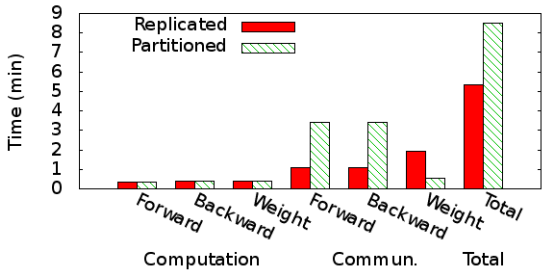

Figure 14: Partitioned and replicated output layer time.

replication, each worker performs all the computations and activation communications of the output layer for half of the images. Thus, partitioning incurs twice the activation communication overheads, while replication incurs twice the weights communications overheads since each worker maintains a copy of the weights.

We measured the impact of partitioning and replicating the output layer of the ImageNet-22K model on Adam with 4-workers (other settings follow the first configuration in Table 1). Figure 14 reports the total time and breakdown of the output layer. The results are consistent with our analysis: partitioning incurs higher communication costs for activations, but lower for weights. Thus, the superior design depends on the DNN and hardware characteristics. For our evaluation environment, our performance model choses replication as the better option, consistent with the measurement results.

Training Time Across the Search Space. Figure 15 presents the training time of ImageNet-22K on different configurations (i.e., parameter servers, replicas and workers) across the search space of using a total of 20 servers in a scatter plot. Each point represents one configuration, and the size of the point indicates the range of its epoch time. The figure shows that (1) there are many configurations choices even with 20 servers only, and without considering different combinations of layer compositions, (2) the training time difference between the best configuration (11.88 min) and the worst configuration (218.08 min) is significant, more than 10x. This gap will grow further with more resources. (3) The optimal solution is surrounded by many suboptimal ones, and not easy to identify. These

results demonstrate the importance of our performance models on designing and configuring distributed DNN systems. The results of MNIST confirm these findings as shown in the tech-report [31].

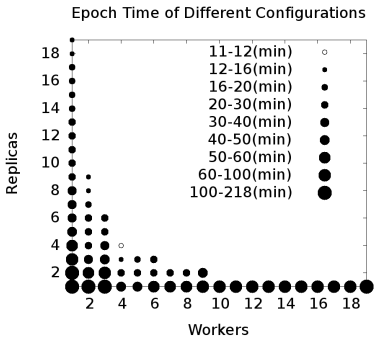

Figure 15: Epoch time across the search space (use 20 machines in total) using ImageNet-22K.

6.5 Scalability on More & Custom Hardware

Another usage scenario of our performance models is to provide scalability estimation for different hardware. We demonstrate it using two cases: (i) scaling out ImageNet-22K on more machines, and (ii) scaling up the worker nodes through custom hardware accelerators of computation, e.g., ASIC and FPGA, and communication, e.g., Infiniband and RDMA. The results are presented in our tech-report [31] in the interest of space, and are consistent with [5].

7. RELATED WORK

Prior work on using parallelism and distributed systems to scale up machine learning algorithms have mostly focused on linear convex problems [1, 26, 27]. Distributed approaches for training large DNNs include using GPU [8] or commodity server [5, 13] clusters. The GPU approach exploits model parallelism to partition the DNN across 16 GPUs connected by Infiniband, while the commodity server approach exploits both data and model parallelism to partition the training data and DNN across $100s$ of machines connected by Ethernet. Our models cover all the design choices supported by these prior studies. Our scalability analysis and optimization tool efficiently configures distributed hardware resources for different learning tasks, improving system scalability, without the human and computational costs of manual system tuning.

Modeling the performance of parallel computing and distributed systems for scalability analysis, resource allocation, and capacity planning is not a new topic; there is a large amount of work in the literature [2, 19, 20, 24, 29]. However, each work is targeted on specific distributed software, hardware systems or applications, such as MapReduce, search engine, stream processing. They apply domain specific information and techniques to achieve desired modeling accuracy and optimization objective. Our work exploits the distinct features and optimization techniques of the distributed DNN systems, which none of the prior work does.

8. CONCLUSION

This paper develops performance models for estimating the scalability of distributed deep learning training, and for driving a scalability optimizer that efficiently determines the optimal system configurations to minimize DNN training time. Experimental results on benchmark DNN tasks validate the estimation accuracy of our models and the effectiveness of the scalability optimizer.

References

[1] A. Agarwal and J. C. Duchi. Distributed delayed stochastic optimization. In *NIPS*, 2011.

[2] S. Babu. Towards automatic optimization of mapreduce programs. In *SoCC*, 2010.

[3] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2009.

[4] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, 2010.

[5] T. Chilimbi, J. Apacible, K. Kalyanaraman, and Y. Suzue. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[6] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *HotOS*, 2013.

[7] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. In *CoRR*, 2010.

[8] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng. Deep learning with cots hpc systems. In *ICML*, 2013.

[9] A. Coates, H. Lee, and A. Ng. An analysis of single-layer networks in unsupervised feature learning. In *AISTATS*, 2011.

[10] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, Nov. 2011.

[11] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Trans. Audio, Speech and Lang. Proc.*, 2012.

[12] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS*, 2014.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[15] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[16] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[17] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck. Learning deep structured semantic models for web search using clickthrough data. In *CIKM*, 2013.

[18] D. Hubel and W. T.N. Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, 1959.

[19] G. Jacques-Silva, Z. Kalbarczyk, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer. Modeling stream processing applications for dependability evaluation. In *DSN*, 2011.

[20] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–603, 2000.

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

[22] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. In *ICML*, 2012.

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[24] Y. Li, P. P. Lee, and J. C. Lui. Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization. In *Sigmetrics*, 2013.

[25] D. P. Mandic and J. Chambers. *Recurrent Neural Networks for Prediction: Learning Algorithms,Architectures and Stability*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[26] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *NAACL HLT*, 2010.

[27] R. Mcdonald, M. Mohri, N. Silberman, D. Walker, and G. S. Mann. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, 2009.

[28] F. Niu, B. Recht, C. Re, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[29] G. Singh, C. Kesselman, and E. Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In *HPDC*, 2007.

[30] U. Vazirani, S. Rao, A. Blanca, and A. Prakash. U.C. berkeley, cs270 algorithms. `http://www.cs.berkeley.edu/~satishr/cs270/sp11/rough-notes/matching.pdf`. Accessed: 2015-02-01.

[31] F. Yan, O. Ruwase, Y. He, and T. Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. `http://www.cs.wm.edu/~fyan/TechReport.pdf`. Accessed: 2015-06-09.