Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters

Paper #57, 16 pages

Abstract

Deep learning workloads are common in today's production clusters due to the proliferation of deep learning driven AI services (*e.g.*, speech recognition, machine translation). A deep learning training job is resource-intensive and time-consuming. Efficient resource scheduling is the key to the maximal performance of a deep learning cluster. Existing cluster schedulers are largely not tailored to deep learning jobs, and typically specifying a fixed amount of resources for each job, prohibiting high resource and job efficiency. This paper proposes *Optimus*, a customized job scheduler for deep learning clusters, which optimizes resource utilization and job performance based on online resource-performance models. *Optimus* uses online fitting to predict model convergence during training, and sets up performance models to accurately estimate training speed as a function of allocated resources in each job. Based on the models, a simple yet effective method is designed and used for dynamically allocating resources and placing deep learning tasks to minimize job completion time. We implement *Optimus* on top of Kubernetes, a cluster manager for container orchestration, and experiment on a deep learning cluster with 7 CPU servers and 6 GPU servers, running 9 training jobs using the MXNet framework. Results show that *Optimus* outperforms representative cluster schedulers by about 139% and 63% in terms of average completion time and makespan, respectively.

1. Introduction

The recent five years have witnessed substantial progress and successful applications of deep learning in various domains of AI, such as computer vision [33], natural language processing [56] and speech recognition [59]. The rising amount of data and increasing scale of training models (*e.g.*, deep neural networks) significantly improve the learning accuracy, as well as remarkably extending the training time. Distributed machine (deep) learning frameworks have been designed and deployed to expedite model convergence using parallel training with multiple machines, *e.g.*, TensorFlow [21], MXNet [50]. Most leading IT companies have been operating distributed machine learning (ML)/deep learning (DL) clusters, with hundreds or thousands of (GPU) servers, to train various ML models over large datasets for their AI-driven services.

Even with parallel training, a deep learning job is resource intensive and time consuming. For example, to train DeepSpeech2 model [22] on LibriSpeech dataset (1000 hours speech) [8], it takes 2–3 days to achieve the state-of-the-art accuracy on 4 servers with 32 GPUs [22]. In a shared deep learning cluster with various training jobs submitted over time, efficient resource scheduling is the key to maximize utilization of expensive resources (*e.g.*, GPUs and RDMA networks) for expedited training completion. However, achieving high training performance and resource efficiency in deep learning clusters is challenging for existing cluster schedulers.

First, schedulers used in existing ML/DL clusters (*e.g.*, Google uses Borg [54] and Microsoft, Tencent, and Baidu use YARN-like schedulers [52] for managing ML jobs) allocate a fixed amount of resource to each job upon its submission, according to resource requirements specified by the job owner. Jobs already running in the cluster cannot benefit from extra resources when they are available (*e.g.*, during night time when there are less workloads), unless the cluster operator manually reconfigures their resource composition or a job owner resubmits the job as new. This may well lead to low resource utilization efficiency.

Second, existing schedulers are designed for different workloads but deep learning. For example, Mesos, Yarn and Borg are for general-purpose cluster resource management; *Corral* [36] is designed for periodic data-parallel jobs, and *TetriSched* [51] handles reservation-based workloads. There is room for improving resource utilization in deep learning clusters with a tailor-made resource scheduler that leverages structures of deep learning frameworks (*e.g.*, the parame-

ter server architecture) and characteristics of deep learning jobs (*e.g.*, iterativeness, convergence properties) for maximal training efficiency.

This paper proposes *Optimus*, a customized cluster scheduler for deep learning training jobs that targets high job performance and resource efficiency in production clusters. We focus on data parallel DL training jobs using the parameter server framework (§2). *Optimus* builds resource-performance models for each job on the go, and dynamically schedules resources to jobs based on job progress and the cluster load to minimize average job completion time and makespan. Specifically, we make the following contributions in developing *Optimus* .

▷ We build accurate performance models for deep learning jobs (§3). Through execution of a training job, we track the training progress on the go and use online fitting to predict the number of steps/epochs required to achieve model convergence (§3.1). We further build a resource-performance model by exploiting communication patterns of the parameter server architecture and iterativeness of the training procedure (§3.2). Different from existing detailed modeling of a distributed deep learning job (such as in [60]), our resource-performance model requires no knowledge about internals of the ML model and hardware configuration of the cluster. The basis is an online learning idea: we run a job for a few steps with different resource configurations, learn the training speed as a function of resource configurations using data collected from these steps, and then keep tuning our model on the go.

▷ Based on the performance models, we design a simple yet effective method for dynamically allocating resources to average minimize job completion time (§4.1). We also propose a task placement scheme for deploying parallel tasks in a job onto the servers, given the job's resource allocation (§ 4.2). The scheme further optimizes training speed by mitigating communication overhead within a job during training.

▷ We discover a load imbalance issue on parameter servers with the existing parameter server framework as in MXNet [50], which significantly lowers the training efficiency. We resolve the issue by assigning model slices to parameter servers evenly (§5.3). We integrate our scheduler *Optimus* with Kubernetes [13], Google's cluster manager for automated container orchestration. The source code of this project is available at `https://github.com/eurosys18-Optimus/Optimus`. We build a deep learning cluster consisting of 7 CPU servers and 6 GPU servers, and run 9 representative DL jobs from different application domains (see Table 1). Evaluation results show that *Optimus* achieves high job performance and resource efficiency, and outperforms widely adopted cluster schedulers by 139% and 63% in average completion time and makespan, respectively (§6).

2. Background and Motivation

2.1 DL Model Training

A deep learning job trains a DL model, such as a deep neural network (DNN), using a large amount of training samples, to minimize a loss function (typically) [41].

Iterativeness. The model training is usually carried out in an iterative fashion, due to the complexity of DNNs (*i.e.*, no closed-form solution) and the large size of sample dataset (*e.g.*, 14 million images in full Imagenet dataset [11]). The dataset is commonly divided into equal-sized data chunks, and each data chunk is further divided into equal-sized *mini-batches*. In each training *step*, we process one mini-batch by computing what changes to be made to the parameters in the DL model to approach their optimal values (typically expressed as gradients, *i.e.*, directions of changes), using data in the mini-batch, and then update parameters using a formula like $new\_parameter = old\_parameter - learning\_rate \times gradient$. A training performance metric is also computed for each mini-batch, *e.g.*, training loss (a summation of the errors made for each example in the mini-batch) or accuracy (the percentage of correct predictions compared to the true data), validation loss or accuracy (computed on validation dataset for model evaluation). After all mini-batches in the dataset have been processed once, one training *epoch* is done.

Convergence. The dataset is usually trained for multiple epochs (tens to hundreds) until the model converges, *i.e.*, the decrease or increase in the performance metric's value between consecutive epochs becomes very small. An illustration of the training curves, the variation of training/validation loss and accuracy vs. the number of training epochs, is given in Fig. 1, for the example of training ResNext-110 [57] on the CIFAR10 dataset [2]. DNN models are usually non-convex and we can not always expect convergence [25]. However, different from experimental models, production models are mature and can typically converge to the global/local optimum very well since all hyper-parameters (*e.g.*, learning rate (*i.e.*, how quickly a DNN adjusts itself), mini-batch size) have been well-tuned during the experimental phase. In this work, we focus on such production models, and leverage their convergence property to estimate a training job's progress towards convergence.

Especially, we use the convergence of training loss to decide completion of a DL job in this work. The DL model converges if the decrease of training loss between two consecutive epochs has consistently fallen below a threshold that the job owner specified, for several epochs. Training loss based training convergence is common in practice [41]. Training/validation accuracy is difficult to be defined in some scenarios where there is no "right answer", *e.g.*, language modeling [5]. Validation loss is usually used to prevent model overfitting, and evaluation on validation dataset is performed only when necessary (*e.g.*, at the end of each

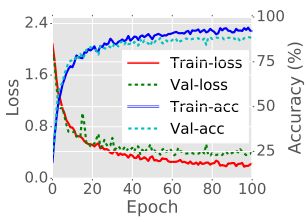


Figure 1: Training curves of ResNext-110 on the CIFAR10 dataset

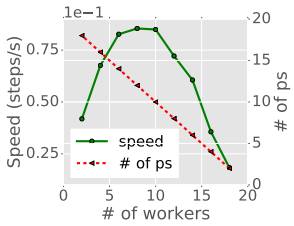


Figure 2: Training speed with 20 parameter servers and workers in total

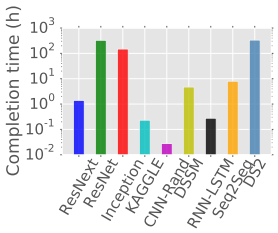


Figure 4: Training time of deep learning models in Table 1

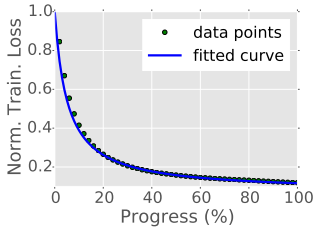


Figure 5: Online model fitting for training Seq2Seq: $\beta_0 = 0.21, \beta_1 = 1.07, \beta_2 = 0.07$

epoch), while we can obtain training loss after each step for more accurate curve fitting (§3.1).

2.2 The Parameter Server Architecture

Most distributed ML frameworks (*e.g.*, MXNet [50], TensorFlow [21], PaddlePaddle [15], Angel [37], Petuum [58]) employ the parameter server (PS) architecture [41] (Fig. 3). In this architecture, the model (*i.e.*, a DNN) is partitioned among multiple parameter servers and the training data are split among workers. Each worker computes parameter updates (*i.e.*, gradients) locally using its data partition and pushes them to parameter servers maintaining the respective model parameters. After receiving gradients, parameter servers update the model parameters using some optimization method, *e.g.*, Stochastic Gradient Descent (SGD) [18] . Updated parameters are sent back to the workers, which then start the next training step, using the updated parameters.

There are two training modes: *asynchronous training*, where the training progress at different workers in a job is not synchronized and each parameter server updates its parameters each time upon receiving gradients from a worker; *synchronous training*, where training progress at all workers is synchronized and a parameter server updates parameters after it has collected gradients from all workers in each step.

2.3 Existing Cluster Schedulers

Static resource allocation. Parameter servers and workers typically run in containers or virtual machines in a DL cluster, and a cluster scheduler manages the resource allocation to training jobs, *e.g.*, Mesos [34] in a TensorFlow cluster [21], Yarn [52] for clusters running MXNet [50] or Angel [37]. With these schedulers, the owner of a training job specifies resource requirements, *e.g.*, the numbers of parameter servers and workers, which remain unchanged throughout the training process.

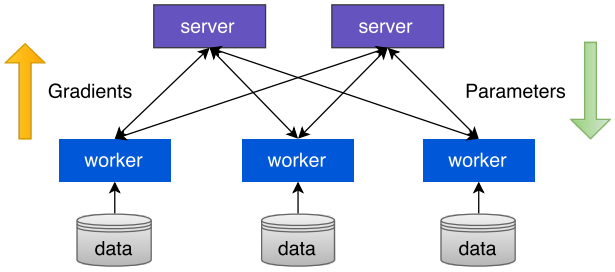


Figure 3: Parameter server architecture

The numbers of workers and parameter servers used to run a training job influence the training speed (*i.e.*, the average number of training steps completed per second), and hence the training completion time significantly. Fig. 2 shows the training speed varies with a different number of workers deployed, when we train a ResNet-50 model [33], one of the state-of-the-art NNs for image classification (details in Table 1), on the ImageNet dataset [11] in a cluster of 20 containers. Each container is configured with 5 CPU cores and 10GB memory, and can run 1 worker or 1 parameter server. If the number of workers is x, then the number of parameter servers used is $20 - x$. In a production cluster, job training speed is further influenced by many runtime factors, such as available bandwidth at the time. Configuring a fixed number of workers/parameter servers upon job submission is hence unfavorable. In *Optimus*, we maximally exploit varying runtime resource availability by adjusting numbers and placement of workers and parameter servers, aiming to pursue the best training speed at each time.

Job size unawareness. Existing schedulers largely adopt FIFO (as in Spark [61]) or Dominant Resource Fairness (DRF) [29] (as in Mesos [34]) as default scheduling strategies, which are ignorant of job sizes (represented by input data size, model complexity, or time taken to complete the job). It has been shown that job performance can be improved by considering job sizes when making scheduling decisions [26]. Training completion time varies significantly among DL jobs. Fig. 4 shows the training time of several representative DL models on respective datasets, as given in Table 1, on a TITAN X GPU. The training time varies from minutes (for simple models on small datasets) to weeks (for complex models on large datasets). *Optimus* takes into account projected job completion time for different jobs when adjusting their resource allocation, to minimize average completion time.

Table 1: Deep learning jobs used for tests and experiments

Model	# of parameters (M)	Network type	Application domain	Dataset	Dataset size
ResNext-110 [57]	8	CNN	image classification	CIFAR10 [2]	60,000
ResNet-50 [33]	25	CNN	image classification	ILSVRC2012-ImageNet [11]	1,313,788
Inception-BN [49]	11.3	CNN	image classification	Caltech [1]	30,607
KAGGLE [12]	1.4	CNN	image identification	Kaggle-NDSB1 [4]	37,920
CNN-rand [40]	6	CNN	sentence classification	MR [23]	10,662
DSSM [46]	1.5	RNN	word representation	text8 [42]	214,288
RNN-LSTM-Dropout [20]	4.7	RNN	language modeling	PTB [17]	1,002,000
Sequence-to-Sequence [28]	4	RNN	machine translation	WMT17 [19]	1,000,000
DeepSpeech2 [22]	38	RNN	speech recognition	LibriSpeech [8]	45,000

3. Performance Modeling of DL Jobs

To make good resource scheduling decisions, we would like to know the relation between resource configuration and the time a training job takes to achieve model convergence. We derive this relation by estimating online how many more training epochs a job needs to run for convergence (§3.1), and how much time a job needs to complete one training epoch given a certain amount of resources (§3.2).

3.1 Learning the Convergence Curve

We draw the training loss curve with the training progress of each DL job, and do online model fitting, in order to predict how far the job is from convergence.

Data preprocessing. For better model fitting, we carry out outlier removal as follows: if a loss data point does not fall within a certain range of its neighbours (*e.g.*, between the minimum loss in its subsequent 5 epochs and the maximum loss in its previous 5 epochs), we consider the data point as an outlier, and use the average value of its neighbours to replace this point when doing model fitting. We also normalize the loss values, by dividing each raw value by the maximum loss value collected so far (typically the first loss value). In this way, loss values in different DL jobs are all between 0 and 1. Fig. 6 shows example loss curves collected by running example DL jobs in Table 1 (which are DL examples from official MXNet tutorials [14]), using the MXNet framework on a server with 1 E5-1650 v4 CPU and 2 NVIDIA TITAN X GPUs. The training progress is ratio of current epoch over the convergence epoch.

Online fitting. We observe that most DL jobs use SGD to update parameters and approximate the optimal parameter values. Since SGD converges at a rate of $O(1/k)$ in terms of the number of steps k, we use the following model to fit the training loss curve:

$$l = \frac{1}{\beta_0 \cdot k + \beta_1} + \beta_2 \quad (1)$$

where l denotes the training loss, and β_0, β_1 and β_2 are positive coefficients. Our online model fitting is carried out as follows: after each training step, we collect a training loss data point (k, l); we then preprocess the data as described above and use a non-negative least squares (NNLS) solver [6] to find the best coefficients that fit the loss points collected so far. In some cases hundreds of thousands of steps are needed to achieve model convergence; in such a case we can sample loss data every few steps, or average the values of several data points (*e.g.*, all losses in an epoch) as a single data point, to reduce the number of data points fed into the solver. Since we can collect more and more loss data as the job runs, the fitted model improves continuously. An example of model fitting when training the Seq2Seq model in Table 1 is given in Fig. 5.

At each step, using the fitted loss model and the predefined convergence threshold δ, we can easily calculate the total number of steps/epochs a job needs to achieve convergence, as well as the number of steps/epochs left from now on until convergence. With more and more data points collected for model fitting, the estimation of the total number of steps/epochs a job needs improves gradually, as illustrated in Fig. 7. Here the prediction error is the difference between the estimated total number of epochs for the model training to converge and the actual total number of epochs needed, divided by the actual number.

3.2 Resource-Speed Modeling

We next build a resource-to-training speed model based on computation and communication patterns in a parameter server architecture.

System models. In a typical DL job, the time taken to complete one training step includes the time for doing forward propagation (*i.e.*, loss computation) and backward propagation (*i.e.*, gradients computation) at the worker, the worker pushing gradients to parameter servers, parameter servers computing parameter updates, and the worker pulling updated parameters from parameter servers, as well as due to extra communication overhead. Suppose there are p parameter servers and w workers in the job. The bandwidth capacity of each parameter server is B, and the model size (*i.e.*, total bytes of parameters) is S. The forward propagation time when a worker trains a minibatch is $m \cdot T_{forward}$ (the size of a mini-batch times the average processing time of one sample). The backward propagation time T_{back} is not related to m and is typically fixed. The size of gradients is the same as the model size S. If the parameters are evenly distributed on parameter servers (load balanced as discussed in §5.3), the size of gradients sent between a worker and a parameter

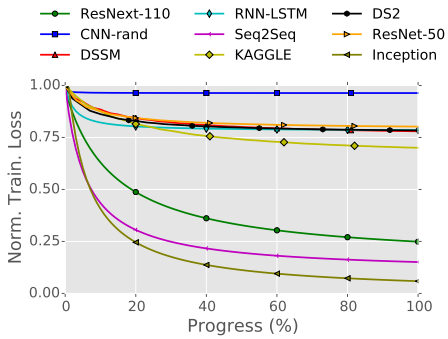


Figure 6: Training loss curves for different DL jobs

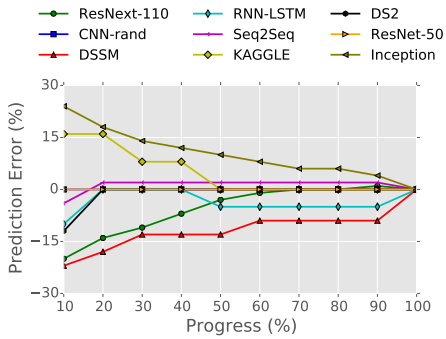


Figure 7: Prediction errors in different DL jobs

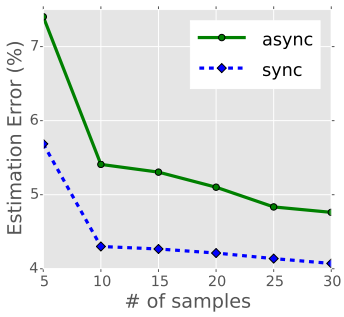


Figure 8: Estimation errors of training speeds

server is $\frac{S}{p}$. In practice, the bandwidth bottleneck between a worker and a parameter server usually lies at the parameter server side. Let w' denote the number of workers that send gradients to a parameter server at the same time. Then the bandwidth between the worker and parameter server ρ is $\frac{B}{w'_\rho}$. Pushing gradients and pulling updated parameters are symmetric processes, so the data transfer time of each worker is $2\frac{S/p}{B/w'_\rho}$. The parameter update time on a parameter server is $\frac{T_{update} \cdot w'_\rho}{p}$ on average, where T_{update} is the time to update parameters with size S. In addition, the communication overhead (*e.g.*, handling TCP connections and control messages between parameter servers and workers) increases linearly with the number of parameter servers and the number of workers. It is represented by $\delta \cdot w + \delta' \cdot p$ where δ and δ' are coefficients.

Therefore, the duration of one training step on a worker can be modeled as

$$T = \max_{\rho}[m \cdot T_{forward} + T_{back} + 2\frac{S/p}{B/w'_\rho} + \frac{T_{update} \cdot w'_\rho}{p} + \delta \cdot w + \delta' \cdot p] \quad (2)$$

According to Eqn. 2, the workers should have similar processing speeds and parameter servers should be load-balanced, in order to achieve minimal time per training step. We will discuss how to handle slow workers in §5.2 and achieve load balancing among parameter servers in §5.3.

We next model the training speed in a job based on Eqn. 2, which is the number of training steps completed per unit time. We divide our models in two cases.

Asynchronous training, where the workers process mini-batches in their own pace. The overall number of training steps completed by all workers per unit time is $w \cdot T^{-1}$. Suppose w'_ρ is linear with w, since more workers may concurrently communicate with one parameter server if the total number of workers is larger. Then the training speed achieved with p parameter servers and w workers can be modeled as

$$f(p, w) = w \cdot (\theta_0 + \theta_1 \cdot \frac{w}{p} + \theta_2 \cdot w + \theta_3 \cdot p)^{-1} \quad (3)$$

where $\boldsymbol{\theta}$ are positive coefficients, corresponding to respective terms in Eqn. 2. We seek to learn the coefficients by fitting the model with runtime data collected for each job.

Synchronous training, where all workers progress from one step to the next in sync. The training speed is T^{-1}. w'_ρ equals w since all workers are synchronized. For synchronous training, the batch size, *i.e.*, the overall size of all mini-batches trained by all workers in each step , needs to remain the same, no matter how we adjust the number of concurrent workers over time. This guarantees that the same training result (model) can be achieved while varying the number of workers [31]. Let M denote the batch size which is typically specified in the training job when the owner submits it. Then the mini-batch size on each worker is $m = \frac{M}{w}$. Then the training speed function can be modeled as

$$f(p, w) = (\theta_0 \cdot \frac{M}{w} + \theta_1 + \theta_2 \cdot \frac{w}{p} + \theta_3 \cdot w + \theta_4 \cdot p)^{-1} \quad (4)$$

where $\boldsymbol{\theta}$ are positive coefficients, to be learned for each job.

Model fitting. To learn the values of θ's and build the training speed functions in Eqn. 3 and Eqn.4, we need to collect data points $(p, w, f(p, w))$. Before we run each training job, we train its model on a small sample set of input data for several steps, with possible combinations of p and w. Each run takes only tens of seconds. In each run, we derive the average training speed under (p, w). Due to the iterative nature of DL model training, training for several steps is enough to give us a good idea of the training speed $f(p, w)$. Then we use NNLS to find θ's that best fit the collected data points $(p, w, f(p, w))$. This initial training speed function constructed is used for resource scheduling decisions when we start running the actual job. Over the training process, we keep collecting data points $(p, w, f(p, w))$ and use them to calibrate coefficients in our training speed models.

Fig. 9 shows the collected data points and fitted trained speed function curves, when we run the ResNet-50 job in a cluster of 40 containers using synchronous training and asynchronous training, respectively. We make three important observations: (a) Our speed function can closely de-

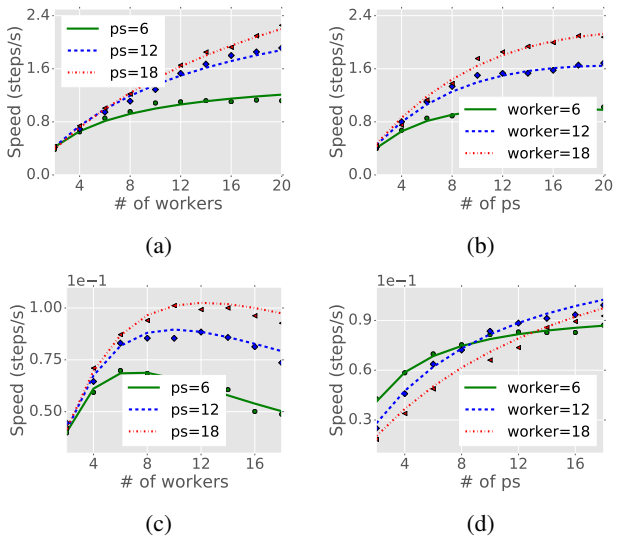


Figure 9: Data points and fitted curves of speed functions for asynchronous training (a)(b) and synchronous training (c)(d)

scribe the relationship between the training speed and resource configurations. (b) Due to communication overhead, there is a trend of diminishing return where adding more parameter servers or workers does not improve the training speed much. (c) For synchronous training, more workers may lead to lower training speed. This is because more workers lead to smaller mini-batch size $\frac{M}{w}$ (*i.e.*, a lower workload on each worker), which may cause CPU/GPU underutilization. Meanwhile, a larger number of workers lead to higher synchronization cost and communication overhead.

Table 2 lists derived coefficients in the speed functions for asynchronous and synchronous training, respectively. We find that forward propagation, backward propagation and data transfer make up most of the training time of one step, since coefficients of these quantities are relatively large.

The reason why we produce the initial training speed function under possible combinations of p and w before running the actual job, is the following: (p, w) pairs used in actual resource configuration when running each job are limited; training speed functions learned using the limited data points may be biased, diverting resource allocation decisions away from the optimum. One question is how many possible (p, w) pairs we should try out to initialize the speed function, to achieve high model accuracy. For the above ResNet-50 example, there are 400 possible (p, w) pairs. Fig. 8 shows the estimation errors of training speeds when we randomly

Table 2: Coefficients in speed functions

	θ_1	θ_2	θ_3	θ_4	θ_5	residual sum of squares for fitting
Async	2.83	3.92	0.00	0.11	-	0.10
Sync	1.02	2.78	4.92	0.00	0.02	0.00

select a number of samples, *i.e.*, (p, w) pairs, to produce the training speed function. The estimation error is the absolute ratio of the gap (between the measured speed and the estimated speed) over the measured speed. We observe that: (a) we can get a less than 6% error even when we only use 10 (p, w) pairs to learn the speed function; (b) using data points of more (p, w) pairs leads to smaller error, but with a diminishing return.

4. Dynamic Scheduling

In our DL cluster, jobs arrive in an online manner. *Optimus* periodically allocates resources to the active jobs (new jobs submitted in the previous scheduling interval and unfinished jobs submitted earlier), by adjusting the numbers and placement of parameter servers/workers in each job in the shared DL cluster. Its scheduling algorithm consists of two parts: resource allocation and task placement.

4.1 Resource Allocation

In each scheduling interval, let Q_j denote the remaining number of epochs that a job j needs to run to achieve model convergence (§3.1), and $f(p_j, w_j)$ be the current training speed function for job j (§3.2). We can estimate the remaining running time t_j of job j as $\frac{Q_j}{f(p_j, w_j)}$. Let O_j^r (N_j^r) denote the amount of type-r resource each worker (parameter server) in job j occupies. C_r is the overall capacity of type-r in the DL cluster and R is the number of resource types. J is the set of current active jobs. Our scheduler aims to minimize the average completion time of these jobs. We can solve the following optimization problem to decide the numbers of workers/parameter servers for each job $j \in J$, where (7) is the capacity constraint:

$$\text{minimize} \quad \sum_{j \in J} t_j \tag{5}$$

$$\text{subject to:} \quad t_j = \frac{Q_j}{f(p_j, w_j)} \qquad \forall j \in J \tag{6}$$

$$\sum_{j \in J} (w_j \cdot O_j^r + p_j \cdot N_j^r) \leq C_r \qquad \forall r \in R \tag{7}$$

$$p_j \in \boldsymbol{Z}^+, w_j \in \boldsymbol{Z}^+ \qquad \forall j \in J \tag{8}$$

The problem is non-convex integer programming problem, which is NP-hard in general. We design an efficient heuristic to solve it (see Function ALLOC in Algorithm 1). We define the *marginal gain* in job completion time reduction as follows:

$$\max\{(\frac{Q_j}{f(p_j, w_j)} - \frac{Q_j}{f(p_j + 1, w_j)})/N_j^D, \\ (\frac{Q_j}{f(p_j, w_j)} - \frac{Q_j}{f(p_j, w_j + 1)})/O_j^{D'}\} \tag{9}$$

Here D (D') is the dominant resource of workers (parameter servers) in job j. A dominant resource is the type of resource that has the maximal share in the overall capacity of the cluster, among all resources used by a worker (parameter

server) [29]. $\frac{Q_j}{f(p_j,w_j)} - \frac{Q_j}{f(p_j,w_j+1)}$ ($\frac{Q_j}{f(p_j,w_j)} - \frac{Q_j}{f(p_j+1,w_j)}$) is the reduction in job completion time when one worker (parameter server) is added to job i; dividing it by the amount of dominate resource that a worker (parameter server) occupies, we obtain the marginal gain per unit dominant resource consumption.

Our resource allocation algorithm in each scheduling interval works as follows. We first allocate one worker and one parameter server to each active job to avoid starvation, and then sort all jobs in order of their marginal gains computed using (9). Then we iteratively select the job with the largest marginal gain and add one worker or parameter server to the job, according to which of the two terms in (9) is larger (*i.e.*, whether adding a worker or parameter server brings larger marginal gain). Marginal gains of the jobs are updated when their resource allocation changes. The procedure repeats until some resource in the cluster is used up, or marginal gains of all jobs become non-positive.

The algorithm makes use of predictions based on online fitted models in §3. To mitigate its performance degration due to prediction errors, we further adopt the following.

(1) *Reducing impact of inaccurate Q_j.* Suppose δ is the training loss threshold (*e.g.*, 1%) to tell model converges in a job . Instead of using δ as the convergence threshold, we initially use a larger threshold δ' (*e.g.*, 3%) for estimating how many epochs the job needs to run to approach it. Using δ', model training is closer to convergence and the estimated number of epochs is more accurate. When δ' has been reached, we continue training and estimate the remaining number of epochs needed to reach the original threshold δ.

(2) *Mitigating influence of inaccurate t_j.* As suggested in [26], overestimate of job size has less influence on job scheduling performance than underestimate. The intuition is that overestimate only influences scheduling of the job itself while underestimate may lead to blocking of multiple waiting jobs. Since estimation errors of Q_j and $f(p_j, w_j)$ are larger at the beginning of a job , we overestimate t_j a bit then by multiplying computed value in (6) by a factor (*e.g.*, 1.05).

4.2 Task Placement

In our model of the training step duration in Eqn. 2, processing time on workers and parameter servers are fixed. We can reduce the time, a.k.a improve training speed, by reducing the time spent on parameters/gradients exchange among workers and parameter servers, which is mainly decided by their placement on different servers in the cluster.

To understand how placement affects the training speed, consider a cluster with 3 servers and a synchronous training job using 2 parameter servers and 4 workers. Each server can host 3 parameter servers or workers. The bandwidth at each parameter server or worker is 1. The size of gradients/parameters transferred between a parameter server and a worker in one training step is 1. Fig. 10 illustrate 3 possible ways for placing the workers/parameter servers. With placement (a), the 2 parameter servers and 4 workers need to transfer $3, 3, 1, 1, 2, 2$ amount of data across servers, respectively. Since the bandwidth between a parameter server and a worker is determined by capacity at both ends and the

Algorithm 1 Job Scheduling Algorithm

- $util\_queue$: a descending queue that sorts jobs based on its utility
- $node\_queue$: a descending queue that sorts nodes based on its available resources

function UPDATE_UTIL(job) ▷ compute job utility
  calculate ps_util
  calculate $worker\_util$
  if $ps\_util > worker\_util$ **then**
    return $(ps\_util, \text{“}ps\text{”})$
  else
    return $(worker\_util, \text{“}worker\text{”})$

function ALLOC($jobs$) ▷ resource allocation
  for $job \in jobs$ **do** ▷ initialization
    Assign 1 ps and 1 worker and enqueue job into $util\_queue$
  while cluster resources are enough && not $util\_queue.empty()$ **do**
    $util, task, job \leftarrow util\_queue.pop()$
    if $util <= 0$ **then**
      break
    if $task ==$ “ps” **then** ▷ allocate one more ps
      $job.num\_ps \leftarrow job.num\_ps + 1$
    else ▷ allocate one more worker
      $job.num\_worker \leftarrow job.num\_worker + 1$
    update cluster resources
    $job.util, task \leftarrow$ UPDATE_UTIL(job)
    $util\_queue.push(job.util, task, job)$

function PLACE($jobs$) ▷ task placement
  for $job \in jobs$ **do**
    $succ\_flag \leftarrow False$ ▷ successful placement
    $cand\_nodes \leftarrow \emptyset$ ▷ candidate nodes
    while not $node\_queue.empty()$ **do**
      $node \leftarrow node\_queue.pop()$
      $cand\_nodes \leftarrow cand\_nodes \cup node$
      if $cand\_nodes$ have enough resources **then**
        place job according to Theorem 1
        update resources on $cand\_nodes$
        $node\_queue.push(cand\_nodes)$
        $succ\_flag \leftarrow True$
        break
    if not $succ\_flag$ **then** ▷ no enough resources
      break

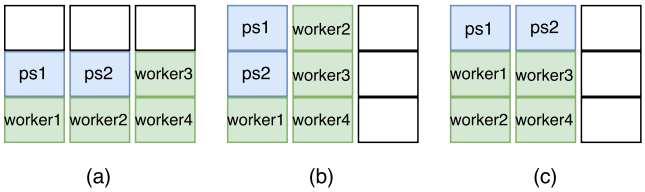


Figure 10: An example of worker/parameter server placement: (c) is best

time is decided by the slowest transfer, we can obtain that the data transfer time in one training step is 3. Similarly, the data transfer time with placement (b) is 3 and with placement (c) is 2. Therefore, in this example, placement (c) is the best solution.

THEOREM 1. *Given the numbers of workers and parameter servers in a job, the optimal worker/parameter server placement principle to achieve the maximal training speed for the job, in a cluster of homogeneous servers, is to use the smallest number of servers to host the job, such that the same number of parameter servers and the same number of workers are deployed on each of these servers.*

The proof is given in the Appendix. Based on this principle, we design a placement scheme (Function PLACE in Algorithm 1) to minimize the data transfer time during training as follows. We sort all servers in the cluster in descending order of their current resource availability (available CPU capacity is used in our experiments). We place jobs in increasing order of their resource demand (*i.e.*, smallest job first). For each job, we check whether the resources on the first k servers are sufficient to host the job (starting with $k = 1$). If so, we place parameter servers and workers in the job evenly on the k servers; otherwise, we check the first $k+1, k+2, \cdots$ servers until we find enough servers to place the job. We then update available resources on the k servers and sort the server list again. The above procedure repeats until all jobs are placed or no sufficient resources on the servers are left to host more jobs. Note that the number of jobs the servers can accommodate might be smaller than the number of jobs we allocate resource to through the resource allocation algorithm (which considers overall resource capacity in the entire cluster). Jobs which are not placed will be temporarily paused and rescheduled in the next scheduling interval.

5. System Implementation

We next present some implementation details of *Optimus*.

5.1 Data Serving

We store training data in Hadoop Distributed File System (HDFS) [3] with a default chunk size of 128M and a replication factor of 2. At the beginning of a job, we assign a roughly equal number of chunks to each worker in a round-robin manner, so that each worker has a similar workload. When the number of workers changes due to our dynamic scaling, we reassign the data chunks so that the workload on each worker is still balanced.

5.2 Straggler Handling

Stragglers, *i.e.*, slow workers (we will discuss the case of slow parameter servers in §5.3), influences a synchronous training job significantly, due to the need of synchronizing all workers in each training step. For asynchronous training, it is also important to ensure the workers have similar training speeds so that the parameters on any worker are not too stale; parameter staleness may lead to unstable training progress and hence additional training steps to achieve convergence [24]. In a distributed DL framework, stragglers may happen due to a number of reasons, *e.g.*, resource contention, unbalanced workload.

To detect stragglers in an asynchronous training job, we simply monitor each worker's training speed: if a worker is too slow (*i.e.*, half speed from the median), we consider it as a straggler. For synchronous training, the training speeds at the workers are the same since they are synchronized. To identify a straggler, we monitor the time of local computation (*i.e.*, forward and backward propagation) and the time of gradients/parameters transfer in some training steps at each worker. If either the computation time or the data transfer time of a worker is much larger than (*i.e.*, twice of) others', we regard it as a straggler. We replace a straggler by launching a new worker.

5.3 Load Balancing on Parameter Servers

Our DL jobs are running on the MXNet framework. We identify possible significant load imbalance among parameter servers in MXNet, due to its way of dividing model parameters among parameter servers: for each block of parameters (*i.e.*, parameters of one layer in a NN), if its size (*i.e.*, the number of parameters) is smaller than a threshold (10^6 by default), then it is assigned to one parameter server randomly; otherwise it is sliced evenly among all parameter servers. Setting the threshold is difficult since different models may have different appropriate thresholds, and different threshold values often lead to a big difference in computation workload among parameter servers. Such a load imbalance problem also exists in other distributed ML frameworks such as TensorFlow.

To balance the workload among parameter servers (mainly due to parameter update computation and communication overhead), we seek to minimize (a) the maximal difference of parameter sizes between two parameter servers, (b) the total number of parameter update requests between parameter servers and workers during one training step (each request from a worker asks for one updated parameter block), and (c) the maximal difference of the number of parameter update requests between two parameter servers. We design a parameter assignment algorithm (Algorithm 2), PAA, as follows.

Algorithm 2 PAA: Parameter Assignment Algorithm

function ASSIGN_PARAMS($block\_arr, num\_ps$)
  Initialize $sizes, transfers$ to **0**
  Initialize map, $block\_queue$ to $\emptyset$
  for $block \in block\_arr$ **do** ▷ sort in descending order
    $block\_queue.push(-block.size, block)$
  $avg\_size \leftarrow sum(block\_arr)/num\_ps$
  while not $block\_queue.empty()$ **do**
    $\_, block \leftarrow block\_queue.get()$
    if $block.size < avg\_size$ **then**
      if $block.size < avg\_size \times 1\%$ **then**
        $ps \leftarrow \text{argmin}_i\, transfers[i]$
      else
        $ps \leftarrow$ BEST_FIT$(avg_size, sizes, block)$
    else ▷ split the block into two partitions
      $ps \leftarrow \text{argmin}_i\, sizes[i]$
      $block.size \leftarrow block.size - \min(block.size, avg\_size)$
      $block\_queue.push(-block.size, block)$
    $sizes[ps] \leftarrow sizes[ps] + \min(block.size, avg\_size)$
    $transfers[ps] \leftarrow transfers[ps] + 1$
    $map[block.key] \leftarrow ps$ ▷ map a block to a ps
  return map

We sort parameter blocks in decreasing order of size and calculate the average parameter size avg_size, *i.e.*, the overall parameter size divided by the number of parameter servers. For each block, if its size is very small (*e.g.*, less than 1% of avg_size), then we assign it to the parameter server with the least number of update requests to balance it on each parameter server.If the block size is between 1% of avg_size and avg_size, we assign the block to the parameter server with the smallest remaining capacity (avg_size minus size of parameters assigned), that can accommodate it (a best-fit approach). If the block size is larger than avg_size, we further slice it into partitions with size avg_size or less (for the last block), and assign the sliced blocks to the parameter server with the smallest number of parameters assigned. Once a parameter block (or partition) is assigned to a parameter server, we add the number of parameter update requests on the server by 1.

5.4 Elastic Training on MXNet

To adjust resource allocation to jobs (*i.e.*, numbers of workers and parameter servers) during training, we adopt a checkpoint-based method. When the number of workers/parameter servers assigned to a job changes, we checkpoint the model parameters and save them to HDFS [3]. Then we restart the job from the checkpoint and redeploy parameter servers and workers based on the scheduling decisions. In practical DL clusters, multiple distributed training frameworks may be used. Our approach is simple and general, and can be easily extended for resource scaling in other frameworks with little code modification.

5.5 Scheduler on Kubernetes

We deploy our scheduler *Optimus* as a normal pod (*i.e.*, a unit of deployment that couples one or more containers tightly) on Kubernetes 1.7 [13], which polls the Kubernetes master to obtain cluster information and job states. For fault-tolerance, we use etcd [9] (*i.e.*, a distributed reliable key-value storage) as a fault-tolerant storage of job states. Kubernetes will automatically restart the scheduler if it fails.

6. Evaluation

6.1 Methodology

Testbed. We built a testbed that consists of 7 CPU servers and 6 GPU servers. Each CPU server has two 8-core Intel E5-2650 CPUs, 80GB memory, two 300GB HDDs and each GPU server has one 8-core Intel E5-1660 CPU, two GeForce 1080Ti GPUs, 48GB memory, one 500GB SSD and one 4TB HDD. They are connected by a 48-port Dell N1548 1GbE switch. We deployed Kubernetes 1.7 [13] and HDFS 2.8 [3] in the cluster.

Simulator. To evaluate *Optimus* at a larger scale of cluster and understand its performance with more parameter choices, we also implemented a discrete-time simulator. The simulator uses the following from the traces collected from our testbed experiments: training losses of each kind of jobs, training speeds under different resource configurations, resource capacities of each server, job configurations (*e.g.*, resource requirements of workers/parameter servers), DL model details (*e.g.*, parameter size).

Workload. Job arrival happens randomly between [0,12000]s. Upon an arrival event, we randomly choose the job among the examples in Table 1 and decide to run it using asynchronous training or synchronous training randomly. We vary the convergence threshold of jobs between 1% and 2%. For jobs training the ResNet-50 model or the DeepSpeech2 model, we downscale their dataset sizes so that the experiment can be finished in a reasonable amount of time, as otherwise each experiment run would last for weeks. We verified that the models still converge with the small datasets. After downscaling, one experiment run takes about 6 hours and we repeat each experiment for 3 times to obtain the average results.

Baselines. We compare *Optimus* with two representative schedulers, implemented on Kubernetes as well: (i) a fairness-based scheduler adopted in many resource managers such as Hadoop [10], Yarn [52] and Mesos [34], which uses Dominant Resource Fairness (DRF) [29] to allocate resources to jobs and dynamically reschedules job resource in each scheduling interval. The workers/parameter servers are placed in a load balancing way, according to the default

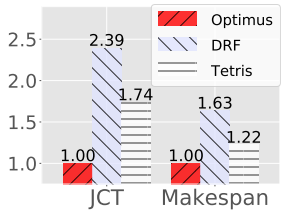


Figure 11: Performance of *Optimus* compared to DRF and Tetris

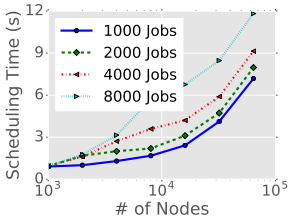
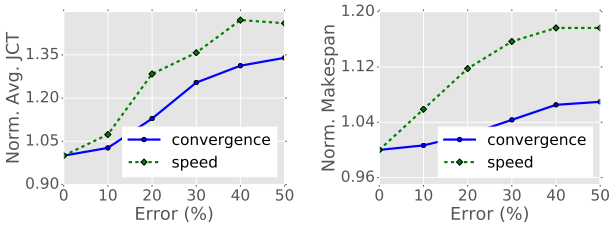


Figure 12: Scalability test



(a) Average completion time (b) Makespan

Figure 14: Sensitivity to prediction errors

behavior of Kubernetes. (ii) Tetris [32], which preferentially allocates resources to jobs with shortest remaining time and packs jobs to servers to minimize resource fragmentation. Since Tetris does not have its own mechanism to estimate the remaining time of a deep learning job, we use our speed function and convergence curve estimation to provide Tetris with such information. We set the ratio of the number of parameter servers to the number of workers to 1:1 [16] in both schedulers.

Metrics. We use the average job completion time (JCT) as an indicator of system performance. In addition, we evaluate the makespan as an indicator of resource efficiency, which is the total time elapsed from the arrival of the first job to the completion of all jobs. Minimizing makespan is equivalent to maximizing resource efficiency [32].

To initialize the training speed function for each job, we prerun a job on a small dataset with 5 different combinations of (p, w). Each scheduling interval is 20 minutes. We set the overestimate factor in §4.1 to 1.05 and set the extremely small size in §5.3 to 1% of avg_size.

6.2 Performance

Comparison with baselines. Fig. 11 shows that *Optimus* can reduce the average completion time and makespan by 2.39x and 1.63x respectively in comparison to the DRF-based fairness scheduler. Fig. 13 shows the number of running tasks and normalized CPU utilization of tasks in each time slot (*i.e.*, CPU utilization divided by overall allocated CPU capacity on a parameter server or a worker) during the whole experiment run. We see *Optimus* does not run a large number of tasks as compared to DRF. The reason is that DRF is work-conserving and allocates as many resources to a job as possible, but more resources do not mean higher training speed, as demonstrated in §3.2. Further, the normalized CPU utilization of workers and parameter servers in *Optimus* is larger than that of DRF and Tetris. It shows that *Optimus* can utilize allocated resources more efficiently.

Resource adjustment overhead. The overhead of changing from one (p, w) configuration to another in a job is measured by the percentage of time spent on adjusting resources of the job. In our experiments, the overall scaling overhead is 2.05% of the makespan, which is acceptable compared to the performance gain.

Scalability. To see whether *Optimus* is sufficiently fast and scalable to large-scale clusters, we emulate submitting and scheduling a large number of jobs in a cluster with thousands of nodes. Figure 12 shows the scheduling time when *Optimus* runs on one core of Intel E5-1620 v4 CPU. *Optimus* can schedule 4,000 jobs (about 100,000 tasks) within 4 seconds on a cluster of 16,000 nodes. This is comparable to the performance of Kubernetes' default scheduler, *i.e.*, 150,000 tasks in 5,000 nodes within 5 seconds [55]. Besides, since *Optimus* makes scheduling decisions at each scheduling interval (*e.g.*, 1 hour), the scheduling overhead is very small.

6.3 Sensitivity analysis

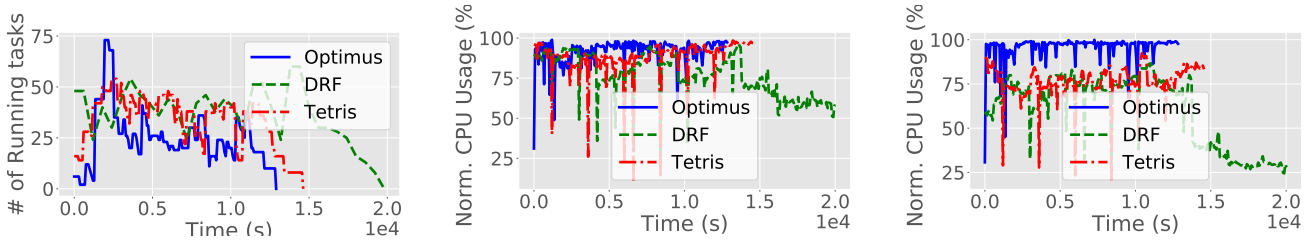
6.3.1 Prediction Error

We examine to what extent *Optimus* is affected by the prediction errors of convergence time and training speed. We carry out simulation under different error levels: suppose the true number of epochs for convergence (training speed) is v and the error is e; we use $v \cdot (1 + e)$ or $v \cdot (1 - e)$ as the input to our scheduler. We run each simulation for 100 times to obtain average results.

In Fig. 14, the convergence (speed) curve plots the resulting average JCT/makespan when we add errors of different levels in convergence epoch (training speed) prediction. When the error is larger, JCT and makespan both increase, but with a diminishing speed. If the error of convergence estimation is 15% and the error of training speed estimation is 6%, there is about 10% performance gap compared to the case where the estimation errors are 0. Compared to the error of convergence estimation, the error of speed estimation affects the performance more. Fortunately, we can estimate training speed much more accurate (6% error) than training convergence (15% error).

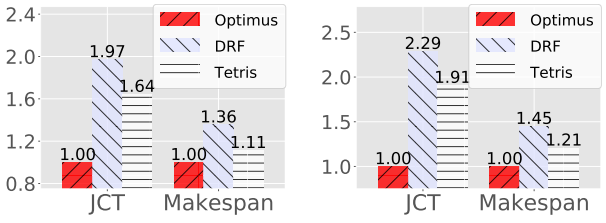
6.3.2 Varying Workloads

Training modes. We examine how training modes affect the performance. Instead of randomly selecting between asynchronous and synchronous training (§6.1), we either train all jobs in asynchronous mode or synchronous mode. Fig. 15



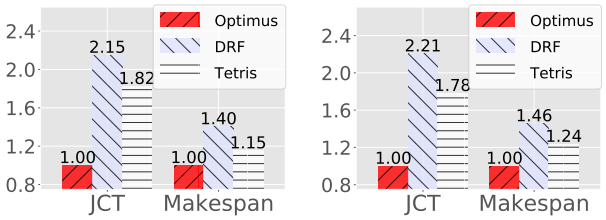
(a) Number of running tasks (b) CPU usage on parameter servers (c) CPU usage on workers

Figure 13: Number of running tasks and CPU usage during an experiment



(a) Async (b) Sync

Figure 15: Sensitivity to workloads: training modes



(a) Poisson (b) Google cluster trace

Figure 16: Sensitivity to workloads: job arrival processes

shows that *Optimus* outperforms the other two schedulers in both cases, and the performance gain is larger when all jobs use synchronous training. This is because all workers have the most updated parameters with synchronous training, such that model convergence is more stable and convergence estimation error is smaller. The training speed of all workers are the same in synchronous training and the speed estimation error is smaller, as verified in §3.2.

Job arrival distributions. We further investigate *Optimus*'s performance under two other job arrival processes. The first is a Poisson process with 3 arrivals per scheduling interval. The second is extracted from Google cluster workload traces over a 7 hour period [7]. Fig. 16 shows that *Optimus* still outperforms the other two schedulers and the performance gain is larger when using Google cluster traces. There are many job arrival spikes in the traces and *Optimus* can handle them better than DRF and Tetris by efficiently allocating resources.

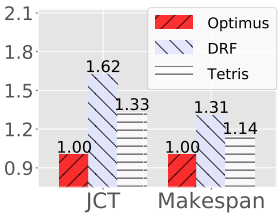


Figure 17: Effectiveness of resource allocation algorithm

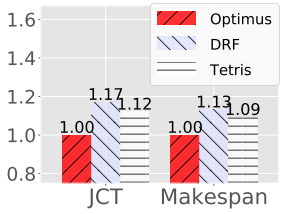


Figure 18: Effectiveness of task placement algorithm

6.4 Inspecting Detailed Designs in *Optimus*

Resource allocation. To see how effective our marginal gain-based resource allocation algorithm is, we replace it with the resource allocation schemes in fairness scheduler or Tetris, while still adopting the same task placement algorithm in *Optimus*. Fig. 17 shows that the average completion time and makespan are reduced by 62% and 31% respectively when using *Optimus* , as compared to the fairness scheduler. That is, the resource allocation algorithm in *Optimus* is critical for high job performance and resource efficiency.

Task placement. We further examine the task placement algorithm in *Optimus* to see to what extent it contributes to job performance and resource efficiency. For comparison, we place tasks using the placement algorithm in fairness scheduler (*i.e.*, in a load-balancing way) and Tetris (*i.e.*, minimizing resource fragmentation), but still use the resource allocation algorithm in *Optimus*. Fig. 18 shows that our algorithm reduces average completion time and makespan by about 10% compared to Tetris and 15% compared to DRF.

Parameter server load balancing. The difference of parameter sizes among parameter servers, the difference of the number of parameter update requests on parameter servers and the total number of update requests between parameter servers and workers are three main factors that represent load imbalance or overhead on parameter servers. Table 3 shows values of the three factors achieved with our PAA algorithm (in §5.3) and with the default parameter distribution algorithm in MXNet, using the ResNet-50 model [33] with 25

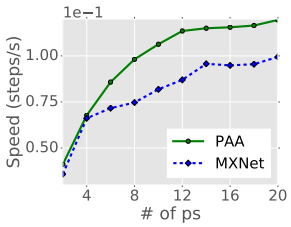


Figure 19: Training speed comparison of ResNet-50 by varying # of ps

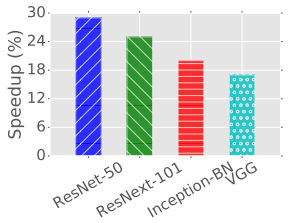


Figure 20: Training speed improvements on different models

million parameters formed into 157 blocks. Our algorithm does not split any parameter block further (since the total number of parameter update requests is 157, the minimal for 157 parameter blocks) while keeping minimal the difference of parameter sizes (*i.e.*, 0.1M) and the difference of the number of requests (*i.e.*, 1).

To see the effectiveness of balanced parameter distribution on the training speed in practice, we train ResNet-50 on the ILSVRC2012 ImageNet dataset [11] by fixing the number of workers to 10 and varying the number of parameter servers, using synchronous training. Fig. 19 shows the training speed with and without our load balancing algorithm. We can see PAA improves the training speed especially when the number of parameter servers is large. Fig. 20 further shows the improvement when more models are trained using synchronous training, 10 workers and 10 parameter servers: PAA achieves up to 30% speedup compared to the MXNet algorithm. We observed similar results with asynchronous training.

In summary, the highlights of our evaluation results are as follows.

(1) Testbed experiments show that *Optimus* improves average job completion time and makespan by 2.39x and 1.63x compared to the fairness scheduler. Further, *Optimus* can scale to schedule 100,000 tasks on 160,000 nodes in 4 seconds, and its resource adjustment overhead is small, *i.e.*, 2.05%.

(2) Further improvement of estimation accuracy will not increase *Optimus*'s performance much (10%) and *Optimus* performs better than DRF and Tetris under various workloads.

(3) The resource allocation algorithm, the task placement scheme and the design of parameter server load balancing contribute to *Optimus*'s improvement by about 62%, 17%, 20% respectively.

Table 3: Comparison of parameter distribution

Algorithm	Difference of parameter size	Difference of # of requests	Total # of requests
MXNet	3.6M	43	247
PAA	0.1M	1	157

7. Discussions

We now discuss extensions and future work on *Optimus*.

Multiple workloads. While *Optimus* targets scheduling of deep learning jobs, it can be used in DL clusters with mixed workloads (*e.g.*, data analytics, online services). For example, in a Kubernetes cluster, we can plug in multiple schedulers and each scheduler is responsible for one kind of workloads. In such case, *Optimus* may ask for resources from a central cluster resource manager and schedule deep learning jobs on a varying portion of cluster resources.

Convergence estimation. For some ML models, the learning rate may be reduced significantly (*e.g.*, by a factor of 0.1) when training reaches a predefined epoch, in order to minimize loss further (*e.g.*, as with SGD). In such a case, we can treat the model training after learning rate adjustment as a new training job and restart online fitting. In addition, the training loss curves of some models (*e.g.*, A3C [43]) cannot be described or can only be partly described using our fitting function in (1), but they may be fitted using other functions [62]. One possible solution is to let the job owner provide the functions, based on the previous running experience of such jobs.

Seamless scaling. We use a checkpoint-based method to adjust the resource configuration of a job due to its simplicity and being generally implementable. This approach may bring quite large overhead if the job has hundreds of workers/parameter servers. Besides, it cannot be applied to models with strong consistency requirements (*i.e.*, making every write to the parameter state durable) [21].

8. Related Work

Performance modeling. Jockey [27] and Morpheus [38] use historical traces of periodic jobs and dynamically adjust resource allocations to meet deadlines, while *Optimus* does not depend on the previous run of the same job since production data change often (*e.g.*, daily). PerfOrator [45] builds a resource-to-performance model of big data queries by estimating query size and profiling hardware, while we use high-level system modeling approach without the knowledge about hardware or the internal details of a job. Ernest [53] runs the entire job on small datasets to estimate the completion time of data analytics. This approach does not work for estimating model convergence, but their sampling method can be used for minimizing sampling overhead when learning the training speed functions. PREDIcT [44] uses sample runs for capturing the convergence trend of a graph algorithm, which is infeasible for deep learning training since the size of dataset affects convergence. Yan *et al.* [60] model the training of deep learning neural networks at a very fine granularity (*e.g.*, the computa-

tion time of each operator on a specific CPU, neural network structures, etc.) while our models capture high-level computation and communication patterns.

Job scheduling. There have been many efforts on cluster/cloud resource allocation to achieve different objectives. Corral [36] and Morpheus [38] focus on periodic or predictable workloads. Borg [54], Fuxi [63], Firmanent [30] are designed for heterogenous workloads in a large-scale cluster and support policy-based scheduling (*e.g.*, fairness, data locality, job priority). Instead, our work focuses on deep learning workload. Mesos [34] and Yarn [52] use DRF [29] to allocate resources while we focus on resource efficiency and job performance. TetriSched [51] and Morpheus [38] also dynamically allocate resources in a global way, but they focus on reservation-based or periodic jobs with specified deadlines. There are several studies [35, 48, 62] on resource allocation of classical machine learning jobs (*e.g.*, clustering, logistic regression) on Spark MLlib [61]. Huang *et al.* [35] propose a memory optimizer for Spark master and workers given a machine learning program. SLAQ [62] targets the training quality of experimental models. Dorm [48] uses a utilization-fairness optimizer to schedule jobs. The main difference is that our work focuses on deep learning jobs running on parameter server architecture. We leverage the characteristics of the jobs to design resource allocation algorithm and task placement scheme, and demonstrate significant performance improvement. STRAD [39] proposes a programming approach to improve model convergence by scheduling parameter updates for model-parallel machine learning, while we did not delve into modifying the underlying ML frameworks.

Distributed machine learning frameworks. The parameter server architecture was first introduced in [47] and improved with update primitives, fault tolerance and communication optimization in [25, 41, 58]. Most distributed machine learning frameworks (*e.g.*, MXNet [50], Petuum [58], TensorFlow [21], Angel [37]) are implicitly or explicitly built upon this architecture. Our work targets scheduling jobs running on these frameworks. We find that the load imbalance problem is common in these distributed frameworks and we propose and implement the PAA algorithm in one of the frameworks, MXNet.

9. Conclusion

Optimus is a customized cluster scheduler targeting high job performance and resource efficiency in deep learning clusters. At its core is an accurate performance model for deep learning workloads, built by exploiting the characteristics of DL model training (*e.g.*, convergence property, iterativeness) and communication patterns of the parameter server architecture. Based on the performance model, we design a marginal gain-based resource allocation algorithm and a training speed-maximizing task placement scheme. Our experiments on a Kubernetes cluster shows *Optimus* outperforms representative cluster schedulers significantly.

Appendix

Proof of Theorem 1

Proof: Assume there are K nodes in the cluster and the number of parameter servers on node k is p_{jk} for job j, and the number of workers on node node k is w_{jk} for job j. Let B_j denote the bandwidth of each parameter server of job j and b_j denote the bandwidth of each worker of job j. Then the data (gradients/parameters) transmission time of job j is

$$\max_k \{\frac{\frac{S_j}{p_j}(w_j - w_{jk})}{B_j}, \frac{\frac{S_j}{p_j}(p_j - p_{jk})}{b_j}\}$$

Then we formulate the problem as follows.

$$\begin{aligned} \text{minimize} \quad & \max_k \{\frac{\frac{S_j}{p_j}(w_j - w_{jk})}{B_j}, \frac{\frac{S_j}{p_j}(p_j - p_{jk})}{b_j}\} \\ \text{subject to:} \quad & \sum_k p_{jk} = p_j \\ & \sum_k w_{jk} = w_j \\ & p_{jk} \in \boldsymbol{Z}^+, w_{jk} \in \boldsymbol{Z}^+ \end{aligned}$$

We decompose the above problem to the following two subproblems whose solutions are guaranteed to be the optimal solution of the above problem. Each subproblem is a lexicographical min max problem whose optimal solution is to place tasks evenly. Combining the optimal solution of the two subproblems, one optimal solution of the original problem is to place parameter servers evenly and place workers evenly on the K nodes. In this proof we ignore node resource constraints since we will always find K nodes to place the job, otherwise the cluster resources are used up and our placement algorithm terminates.

Subproblem 1:

$$\begin{aligned} \text{minimize} \quad & \max_k \frac{\frac{S_j}{p_j}(w_j - w_{jk})}{B_j} \\ \text{subject to:} \quad & \sum_k w_{jk} = w_j \\ & w_{jk} \in \boldsymbol{Z}^+ \end{aligned}$$

Subproblem 2:

$$\begin{aligned} \text{minimize} \quad & \max_k \frac{\frac{S_j}{p_j}(p_j - p_{jk})}{b_j} \\ \text{subject to:} \quad & \sum_k p_{jk} = p_j \\ & p_{jk} \in \boldsymbol{Z}^+ \end{aligned}$$

The next step is to prove that smaller K leads to smaller data transmission time. The proof can be done via mathematical

induction. The intuition is that a smaller K means more parameter servers and workers on each node, so the transferred data amount via network is less and hence the communication time decreases.

References

[1] Caltech 256 Dataset. http://www.vision.caltech.edu/Image_Datasets/Caltech256/, 2006.

[2] The CIFAR-10 Dataset. https://www.cs.toronto.edu/~kriz/cifar.html, 2009.

[3] HDFS. https://wiki.apache.org/hadoop/HDFS, 2014.

[4] Kaggle NDSB1 Dataset. https://www.kaggle.com/c/datasciencebowl/data, 2014.

[5] Perplexity Versus Error Rate. https://nlpers.blogspot.hk/2014/05/perplexity-versus-error-rate-for.html, 2014.

[6] SciPy NNLS. https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.nnls.html, 2014.

[7] Google Cluster Workload Traces. https://github.com/google/cluster-data, 2015.

[8] LibriSpeech ASR Corpus. http://www.openslr.org/12/, 2015.

[9] etcd. https://github.com/coreos/etcd, 2017.

[10] Hadoop CapacityScheduler. https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2017.

[11] ImageNet Dataset. http://www.image-net.org, 2017.

[12] KAGGLE-DSB Model. https://github.com/apache/incubator-mxnet/tree/master/example/kaggle-ndsb1, 2017.

[13] Kubernetes. https://kubernetes.io, 2017.

[14] MXNet Official Examples. https://github.com/apache/incubator-mxnet/tree/master/example, 2017.

[15] PaddlePaddle. http://www.paddlepaddle.org, 2017.

[16] Run Deep Learning with PaddlePaddle on Kubernetes. http://blog.kubernetes.io/2017/02/run-deep-learning-with-paddlepaddle-on-kubernetes.html, 2017.

[17] Penn Tree Bank Dataset. https://catalog.ldc.upenn.edu/ldc99t42, 2017.

[18] Stochastic Gradient Descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent, 2017.

[19] WMT 2017. http://www.statmt.org/wmt17/, 2017.

[20] Word Language Model. https://github.com/apache/incubator-mxnet/tree/master/example/gluon/word_language_model, 2017.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[22] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep Speech 2: End-to-end Speech Recognition in English and Mandarin. In *Proc. of the 33th International Conference on Machine Learning (ICML)*, 2016.

[23] P. Bo and L. Lillian. Movie Review Data. https://www.cs.cornell.edu/people/pabo/movie-review-data/, 2005.

[24] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv preprint arXiv:1604.00981*, April 2016.

[25] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large Scale Distributed Deep Networks. In *Proc. of the 25th Advances in Neural Information Processing Systems (NIPS)*, 2012.

[26] M. Dell'Amico, D. Carra, M. Pastorelli, and P. Michiardi. Revisiting Size-Based Scheduling with Estimated Job Sizes. In *Proc. of IEEE 22th International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2014.

[27] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proc. of the 7th ACM European Conference on Computer Systems (Eurosys)*, 2012.

[28] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional Sequence to Sequence Learning. In *Proc. of the 34th International Conference on Machine Learning (ICML)*, 2017.

[29] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[30] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[31] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. In *arXiv preprint arXiv:1706.02677*, 2017.

[32] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proc. of ACM SIGCOMM*, 2014.

[33] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proc .of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[34] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[35] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource Elasticity for Large-Scale Machine Learning. In *Proc. of ACM SIGMOD*, 2015.

[36] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proc. of ACM SIGCOMM*, 2015.

[37] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui. Angel: a New Large-Scale Machine Learning System. *National Science Review*, page nwx018, 2017.

[38] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[39] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proc. of the 11th European Conference on Computer Systems (Eurosys)*, 2016.

[40] Y. Kim. Convolutional Neural Networks for Sentence Classification. In *Proc. of 19th SIGDAT Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[41] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[42] M. Matt. text8. `http://mattmahoney.net/dc/`, 2017.

[43] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *Proc. of the 33th International Conference on Machine Learning (ICML)*, 2016.

[44] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. PREDIcT: Towards Predicting the Runtime of Large-Scale Iterative Analytics. In *Proc. of the 4th Very Large Data Bases Endowment (PVLDB)*, 2013.

[45] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. PerfOrator: Eloquent Performance Models for Resource Optimization. In *Proc. of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[46] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. A Latent Semantic Model with Convolutional-Pooling Structure for Information Retrieval. In *Proc. of the 23th ACM International Conference on Conference on Information and Knowledge Management (CIKM)*, 2014.

[47] A. Smola and S. Narayanamurthy. An Architecture for Parallel Topic Models. *Proc. of the 1st Very Large Data Bases Endowment (PVLDB)*, 2010.

[48] P. Sun, Y. Wen, T. N. B. Duong, and S. Yan. Towards Distributed Machine Learning in Shared Clusters: A Dynamically-Partitioned Approach. *arXiv preprint arXiv:1704.06738*, 2017.

[49] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. In *Proc. of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[50] C. Tianqi, L. Mu, L. Yutian, L. Min, W. Naiyan, W. Minjie, X. Tianjun, X. Bing, Z. Chiyuan, and Z. Zheng. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Proc. of NIPS Workshop on Machine Learning Systems (LearningSys)*, 2016.

[51] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Proc. of the 11th ACM European Conference on Computer Systems (Eurosys)*, 2016.

[52] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop Yarn: Yet Another Resource Negotiator. In *Proc. of the 4th annual Symposium on Cloud Computing (SoCC)*, 2013.

[53] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proc. of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[54] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proc. of the 10th ACM European Conference on Computer Systems (Eurosys)*, 2015.

[55] T. Wojciech. Kubernetes Scalability. `http://blog.kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html`, 2017.

[56] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation. *arXiv preprint arXiv:1609.08144*, 2016.

[57] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated Residual Transformations for Deep Neural Networks. In *Proc .of the 30th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[58] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *Proc. of the 21th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2015.

[59] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig. The Microsoft 2016 Conversational Speech Recognition System. In *Proc .of the 42th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[60] F. Yan, O. Ruwase, Y. He, and T. Chilimbi. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In *Proc. of the 21th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.

[61] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[62] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *Proc. of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[63] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proc. of the 5th Very Large Data Bases Endowment (PVLDB)*, 2014.