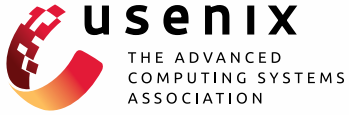

Project Adam: Building an Efficient and Scalable Deep Learning Training System

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman, *Microsoft Research*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Project Adam: Building an Efficient and Scalable Deep Learning Training System

Trishul Chilimbi Yutaka Suzue Johnson Apacible Karthik Kalyanaraman

Microsoft Research

ABSTRACT

Large deep neural network models have recently demonstrated state-of-the-art accuracy on hard visual recognition tasks. Unfortunately such models are extremely time consuming to train and require large amount of compute cycles. We describe the design and implementation of a distributed system called Adam comprised of commodity server machines to train such models that exhibits world-class performance, scaling and task accuracy on visual recognition tasks. Adam achieves high efficiency and scalability through whole system co-design that optimizes and balances workload computation and communication. We exploit asynchrony throughout the system to improve performance and show that it additionally improves the accuracy of trained models. Adam is significantly more efficient and scalable than was previously thought possible and used 30x fewer machines to train a large 2 billion connection model to 2x higher accuracy in comparable time on the ImageNet 22,000 category image classification task than the system that previously held the record for this benchmark. We also show that task accuracy improves with larger models. Our results provide compelling evidence that a distributed systems-driven approach to deep learning using current training algorithms is worth pursuing.

Commodity computing (also known as commodity cluster computing) involves the use of large numbers of already-available computing components for parallel computing, to get the greatest amount of useful computation at low cost.

1. INTRODUCTION

Traditional statistical machine learning operates with a table of data and a prediction goal. The rows of the table correspond to independent observations and the columns correspond to hand crafted features of the underlying data set. Then a variety of machine learning algorithms can be applied to learn a model that maps each data row to a prediction. More importantly, the trained model will also make good predictions for unseen test data that is drawn from a similar distribution as the training data. Figure 1 illustrates this process.

This approach works well for many problems such as recommendation systems where a human domain expert can easily construct a good set of features. Unfortunately it fails for hard AI tasks such as speech recognition or visual object classification where it is extremely hard to construct appropriate features over the input data. Deep learning attempts to address this shortcoming by additionally learning hierarchical features from the raw input data and then using these features to make predictions as illustrated in Figure 2 [1]. While there are a variety of deep models we focus on deep neural networks (DNNs) in this paper.

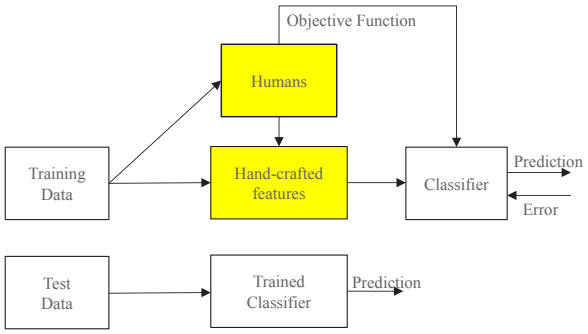

Figure 1. Machine Learning.

Deep learning has recently enjoyed success on speech recognition and visual object recognition tasks primarily because of advances in computing capability for training these models [14, 17, 18]. This is because it is much harder to learn hierarchical features than optimize models for prediction and consequently this process requires significantly more training data and computing power to be successful. While there have been some advances in training deep learning systems, the core algorithms and models are mostly unchanged from the eighties and nineties [2, 9, 11, 19, 25].

Complex tasks require deep models with a large number of parameters that have to be trained. Such large models require significant amount of data for successful training to prevent over-fitting on the

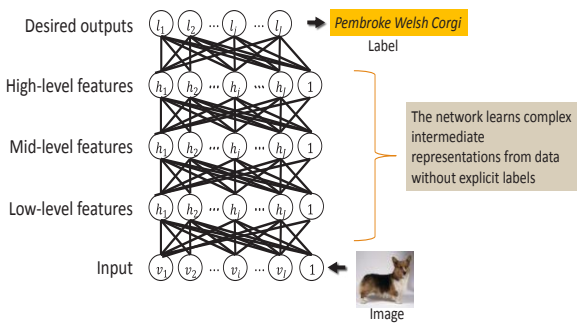

Figure 2. Deep networks learn complex representations.

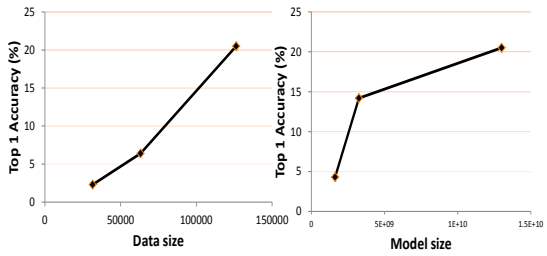

Figure 3. Accuracy improvement with larger models and more data.

training data which leads to poor generalization performance on unseen test data. Figure 3 illustrates the impact of larger DNNs and more training data on the accuracy of a visual image recognition task. Unfortunately, increasing model size and training data, which is necessary for good prediction accuracy on complex tasks, requires significant amount of computing cycles proportional to the product of model size and training data volume as illustrated in Figure 4.

Due to the computational requirements of deep learning almost all deep models are trained on GPUs [5, 17, 27]. While this works well when the model fits within 2-4 GPU cards attached to a single server, it limits the size of models that can be trained. To address this, researchers recently built a large-scale distributed system comprised of commodity servers to train extremely large models to world record accuracy on a hard visual object recognition task—classifying images into one of 22 thousand distinct categories using only raw pixel information [7, 18]. Unfortunately their system scales poorly and is not a viable cost-effective option for training large DNNs [7].

This paper addresses the problem by describing the design and implementation of a scalable distributed deep learning training system called Adam comprised of commodity servers. The main contributions include:

- *Optimizing and balancing both computation and communication for this application through whole system co-design*. We partition large models across machines so as to minimize memory bandwidth and cross-machine communication requirements. We restructure the computation across machines to reduce communication requirements.
- *Achieving high performance and scalability by exploiting the ability of machine learning training to tolerate inconsistencies well*. We use a variety of techniques including multi-threaded model parameter updates without locks, asynchronous batched parameter updates that take advantage of weight updates being associative and commutative, and permit computation over stale parameter values. Surprisingly, it appears that asynchronous training also improves model accuracy.
- *Demonstrating that system efficiency, scaling, and asynchrony all contribute to improvements in trained model accuracy*. Adam uses 30x fewer machines to train a large 2 billion connection model to 2x higher accuracy in comparable time on the ImageNet 22,000 category image classification task than the system that previously held the record for this benchmark. We also show that task accuracy improves with model size and Adam's efficiency enables training larger models with the same amount of resources.

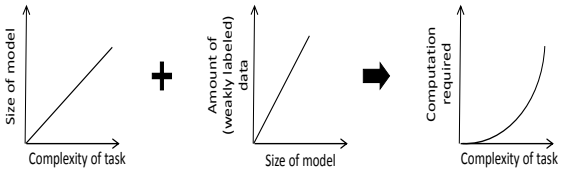

Figure 4. Deep Learning Computational Requirements.

Our results suggest an opportunity for a distributed-systems driven approach to large-scale deep learning where prediction accuracy is increased by training larger models on vast amounts of data using efficient and scalable compute clusters rather than relying solely on algorithmic breakthroughs from the machine learning community.

The rest of the paper is organized as follows. Section 2 covers background material on training deep neural networks for vision tasks and provides a brief overview of large-scale distributed training. Section 3 describes the Adam design and implementation focusing on the computation and communication optimizations, and use of asynchrony, that improve system efficiency and scaling. Section 4 evaluates the efficiency and scalability of Adam as well as the accuracy of the models that it trains. Finally, Section 5 covers related work.

2. BACKGROUND

2.1 Deep Neural Networks for Vision

Artificial neural networks consist of large numbers of homogeneous computing units called neurons with multiple inputs and a single output. These are typically connected in a layer-wise manner with the output of neurons in layer *l-1* connected to all neurons in layer *l* as in Figure 2. Deep neural networks have multiple layers that enable hierarchical feature learning.

The output of a neuron i in layer l, called the activation, is computed as a function of its inputs as follows:

$$a_i(l) = F((\Sigma_{j=1..k} w_{ij}(l\text{-}1,l)*a_j(l\text{-}1)) + b_i)$$

where w_{ij} is the weight associated with the connection between neurons i and j 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.

Convolutional neural networks are a class of neural networks that are biologically inspired by early work on the visual cortex [15, 19]. Neurons in a layer are only connected to spatially local neurons in the next layer modeling local visual receptive fields. In addition, these connections share weights which allows for feature detection regardless of position in the visual field. The weight sharing also reduces the number of free parameters that must be learned and consequently these models are easier to train compared to similar size networks where neurons in a layer are fully connected to all neuron in the next layer. A convolutional layer is often followed by a max-pooling layer that performs a type of nonlinear down-sampling by outputting the maximum value from non-overlapping sub-regions. This provides the network with robustness to small translations in the input as the max-pooling layer will produce the same value.

The last layer of a neural network that performs multiclass classification often implements the softmax function. This function transforms an n-dimensional vector of arbitrary real values to an n-dimensional vector of values in the range between zero and one such that these component values sum to one.

We focus on visual tasks because these likely require the largest scale neural networks given that roughly one third of the human cortex is devoted to vision. Recent work has demonstrated that deep neural networks comprised of 5 convolutional layers for learning visual features followed by 3 fully connected layers for combining these learned features to make a classification decision achieves state-of-the-art performance on visual object recognition tasks [17, 27].

2.2 Neural Network Training

Neural networks are typically trained by back-propagation using gradient descent. Stochastic gradient descent is a variant that is often used for scalable training as it requires less cross-machine communication [2]. In stochastic gradient descent the training inputs are processed in a random order. The inputs are processed one at a time with the following steps performed for each input to update the model weights.

Feed-forward evaluation:

The output of each neuron i in a layer l, called its activation, a, is computed as a function of its k inputs from neurons in the preceding layer $l\text{-}1$ (or input data for the first layer). If $w_{ij}(l\text{-}1,l)$ is the weight associated with a connection between neuron j in layer $l\text{-}1$ and neuron i in layer l:

$$a_i(l) = F((\Sigma_{j=1..k} w_{ij}(l\text{-}1,l)*a_j(l\text{-}1)) + b_i)$$

where b is a bias term for the neuron.

Back-propagation:

Error terms, δ, are computed for each neuron, i, in the output layer, l_n, first as follows:

$$\delta_i(l_n) = (t_i(l_n) - a_i(l_n))*F'(a_i(l_n))$$

where $t(x)$ is the true value of the output and $F'(x)$ is the derivative of $F(x)$.

These error terms are then back-propagated for each neuron i in layer l connected to m neurons in layer $l+1$ as follows:

$$\delta_i(l) = (\Sigma_{j=1..m} \delta_j(l+1)*w_{ji}(l,l+1))*F'(a_i(l))$$

Weight updates:

These error terms are used to update the weights (and biases similarly) as follows:

$$\Delta w_{ij}(l\text{-}1,l) = \alpha*\delta_i(l)*a_j(l\text{-}1) \text{ for j = 1 .. k}$$

where α is the learning rate parameter. This process is repeated for each input until the entire training dataset

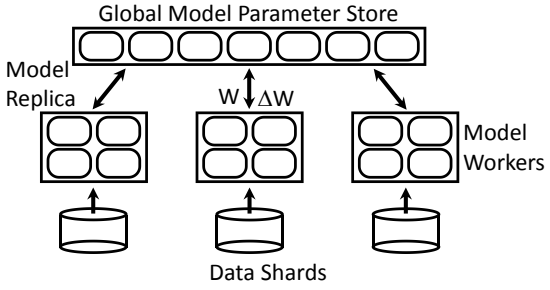

Figure 5. Distributed Training System Architecture.

has been processed, which constitutes a training epoch. At the end of a training epoch, the model prediction error is computed on a held out validation set. Typically, training continues for multiple epochs, reprocessing the training data set each time, until the validation set error converges to a desired (low) value.

The trained model is then evaluated on (unseen) test data.

2.3 Distributed Deep Learning Training

Recently, Dean et al. described a large-scale distributed system comprised of tens of thousands of CPU cores for training large deep neural networks [7]. The system architecture they used (shown in Figure 5) is based on the Multi-Spert system and exploits both model and data parallelism [9]. Large models are partitioned across multiple model worker machines enabling the model computation to proceed in parallel. Large models require significant amounts of data for training so the systems allows multiple replicas of the same model to be trained in parallel on different partitions of the training data set. All the model replicas share a common set of parameters that is stored on a global parameter server. For speed of operation each model replica operates in parallel and asynchronously publishes model weight updates to and receives updated parameter weights from the parameter server. While these asynchronous updates result in inconsistencies in the shared model parameters, neural networks are a resilient learning architecture and they demonstrated successful training of large models to world-record accuracy on a visual object recognition task [18].

3. ADAM SYSTEM ARCHITECTURE

Our high-level system architecture is also based on the Multi-Spert system and consists of data serving machines that provide training input to model training machines organized as multiple replicas that asynchronously update a shared model via a global parameter server. While describing the design and implementation of Adam we focus on the computation and communication optimizations that improve system efficiency and scaling. These optimizations were motivated by our past experience building large-scale distributed systems and by profiling and iteratively improving the Adam system. In addition, the system is built from the ground up to support asynchronous training.

While we focus on vision tasks in this paper, the Adam system is general-purpose as stochastic gradient descent is a generic training algorithm that can train any DNN via back-propagation. In addition, Adam supports training any combination of stacked convolutional and fully-connected network layers and can be used to train models on tasks such as speech recognition and text processing.

3.1 Fast Data Serving

Training large DNNs requires vast quantities of training data (10-100 TBs). Even with large quantities of training data these DNNs require data transformations to avoid over-fitting when iterating through the data set multiple times. We configure a small set of machines as data serving machines to offload the computational requirements of these transformations from the model training machines and ensure high throughput data delivery.

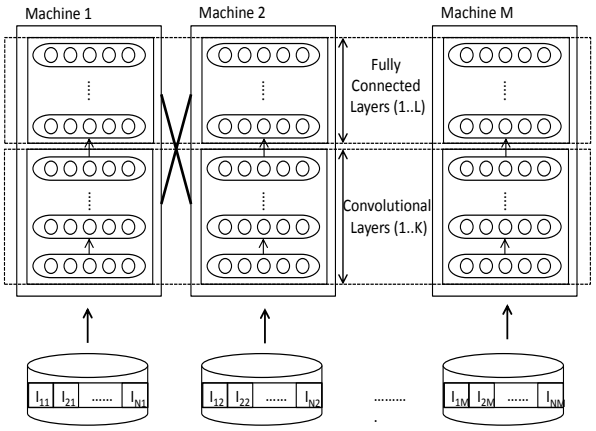

Figure 6. Model partitioning across training machines.

For vision tasks, the transformations include image translations, reflections, and rotations. The training data set is augmented by randomly applying a different transformation to each image so that each training epoch effectively processes a different variant of the same image. This is done in advance since some of the image transformations are compute intensive and we want to immediately stream the transformed images to the model training machines when requested.

The data servers pre-cache images utilizing nearly the entire system memory as an image cache to speed image serving. They use asynchronous IO to process incoming requests. The model training machines request images in advance in batches using a background thread so that the main training threads always have the required image data in memory.

3.2 Model Training

Models for vision tasks typically contain a number of convolutional layers followed by a few fully connected layers [17, 27]. We partition our models vertically across the model worker machines as shown in Figure 6 as this minimizes the amount of cross-machine communication that is required for the convolution layers.

3.2.1 Multi-Threaded Training

Model training on a machine is multi-threaded with different images assigned to threads that share the model weights. Each thread allocates a training context for feed-forward evaluation and back propagation. This

training context stores the activations and weight update values computed during back-propagation for each layer. The context is pre-allocated to avoid heap locks while training. Both the context and per-thread scratch buffer for intermediate results use NUMA-aware allocations to reduce cross-memory bus traffic as these structures are frequently accessed.

3.2.2 *Fast Weight Updates*

To further accelerate training we access and update the shared model weights locally without using locks. Each thread computes weight updates and updates the shared model weights. This introduces some races as well as potentially modifying weights based on stale weight values that were used to compute the weight updates but have since been changed by other threads. We are still able to train models to convergence despite this since the weight updates are associative and commutative and because neural networks are resilient and can overcome the small amount of noise that this introduces. Updating weights without locking is similar to the Hogwild system except that we rely on weight updates being associative and commutative instead of requiring that the models be sparse to minimize conflicts [23]. This optimization is important for achieving good scaling when using multiple threads on a single machine.

3.2.3 *Reducing Memory Copies*

During model training data values need to be communicated across neuron layers. Since the model is partitioned across multiple machines some of this communication is non local. We use a uniform optimized interface to accelerate this communication. Rather than copy data values we pass a pointer to the relevant block of neurons whose outputs need communication avoiding expensive memory copies. For non-local communication, we built our own network library on top of the Windows socket API with IO completion ports. This library is compatible with our data transfer mechanism and accepts a pointer to a block of neurons whose output values need to be communicated across the network. We exploit knowledge about the static model partitioning across machines to optimize communication and use reference counting to ensure safety in the presence of asynchronous network IO. These optimizations reduce the memory bandwidth and CPU requirements for model training and are important for achieving good performance when a model is partitioned across machines.

3.2.4 *Memory System Optimizations*

We partition models across multiple machines such that the working sets for the model layers fit in the L3 cache. The L3 cache has higher bandwidth than main memory and allows us to maximize usage of the floating point units on the machine that would otherwise be limited by memory bandwidth.

We also optimize our computation for cache locality. The forward evaluation and back-propagation computation have competing locality requirements in terms of preferring a row major or column major layout for the layer weight matrix. To address this we created two custom hand-tuned assembly kernels that appropriately pack and block the data such that the vector units are fully utilized for the matrix multiply operations. These optimizations enable maximal utilization of the floating point units on a machine.

3.2.5 *Mitigating the Impact of Slow Machines*

In any large computing cluster there will always be a variance in speed between machines even when all share the same hardware configuration. While we have designed the model training to be mostly asynchronous to mitigate this, there are two places where this speed variance has an impact. First, since the model is partitioned across multiple machines the speed of processing an image is limited by slow machines. To avoid stalling threads on faster machines that are waiting for data values to arrive from slower machines, we allow threads to process multiple images in parallel. We use a dataflow framework to trigger progress on individual images based on arrival of data from remote machines. The second place where this speed variance manifests is at the end of an epoch. This is because we need to wait for all training images to be processed to compute the model prediction error on the validation data set and determine whether an additional training epoch is necessary. To address this, we implemented the simple solution of ending an epoch whenever a specified fraction of the images are completely processed. We ensure that the same set of images are not skipped each epoch by randomizing the image processing order for each epoch. We have empirically determined that waiting for 75% of the model replicas to complete processing all their images before declaring the training epoch complete can speed training by up to 20% with no impact on the trained model's prediction accuracy. An alternative solution that we did not implement is to have the faster machines steal work from the slower ones. However, since our current approach does not affect model accuracy this is unlikely to outperform it.

3.2.6 *Parameter Server Communication*

We have implemented two different communication protocols for updating parameter weights. The first version locally computes and accumulates the weight updates in a buffer that is periodically sent to the parameter server machines when k (which is typically in the hundreds) images have been processed. The parameter server machines then directly apply these

A Level 3 (L3) cache is a specialized cache that that is used by the CPU and is usually built onto the motherboard and, in certain special processors, within the CPU module itself. It works together with the L1 and L2 cache to improve computer performance by preventing bottlenecks due to the fetch and execute cycle taking too long. The L3 cache feeds information to the L2 cache, which then forwards information to the L1 cache. Typically, its memory performance is slower compared to L2 cache, but is still faster than the main memory (RAM).

accumulated updates to the stored weights. This works well for the convolutional layers since the volume of weights is low due to weight sharing. For the fully connected layers that have many more weights we use a different protocol to minimize communication traffic between the model training and parameter server machines. Rather than directly send the weight updates we send the activation and error gradient vectors to the parameter server machines where the matrix multiply can be performed locally to compute and apply the weight updates. This significantly reduces the communication traffic volume from $M*N$ to $k*(M+N)$ and greatly improves system scalability. In addition, it has an additional beneficial aspect as it offloads computation from the model training machines where the CPU is heavily utilized to the parameter server machines where the CPU is underutilized resulting in a better balanced system.

3.3 Global Parameter Server

The parameter server is in constant communication with the model training machines receiving updates to model parameters and sending the current weight values. The rate of updates is far too high for the parameter server to be modeled as a conventional distributed key value store. The architecture of a parameter server node is shown in Figure 7.

3.3.1 Throughput Optimizations

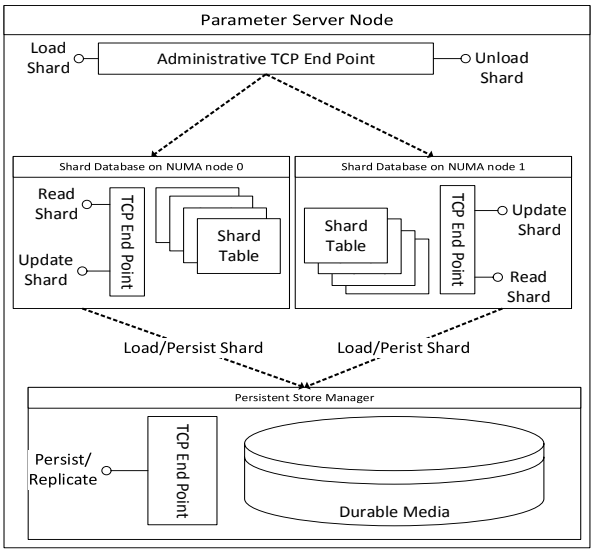

Figure 7. Parameter Server Node Architecture.

The model parameters are divided into 1 MB sized shards, which represents a contiguous partition of the parameter space, and these shards are hashed into storage buckets that are distributed equally among the parameter server machines. This partitioning improves the spatial locality of update processing while the distribution helps with load balancing. Further, we opportunistically batch updates. This improves temporal locality and relieves pressure on the L3 cache by applying all updates in a batch to a block of parameters before moving to next block in the shard. The parameter servers use SSE/AVX instructions for applying the update and all processing is NUMA aware. Shards are allocated on a specific NUMA node and all update processing for the shard is localized to that NUMA node by assigning tasks to threads bound to the processors for the NUMA node by setting the appropriate processor masks. We use lock free data structures for queues and hash tables in high traffic execution paths to speed up network, update, and disk IO processing. In addition, we implement lock free memory allocation where buffers are allocated from pools of specified size that vary in powers of 2 from 4KB all the way to 32MB. Small object allocations are satisfied by our global lock free pool for the object. All of these optimizations are critical to achieving good system scalability and were arrived at through iterative system refinement to eliminate scalability bottlenecks.

空间局限性
批量更新
时间局限性
非均匀访存模型
处理掩码
lock free data structures threads can access this data structure concurrently.Threads canno do same operation

3.3.2 Delayed Persistence

We decouple durability from the update processing path to allow for high throughput serving to training nodes. Parameter storage is modelled as a write back cache, with dirty chunks flushed asynchronously in the background. The window of potential data loss is a function of the IO throughput supported by the storage layer. This is tolerable due to the resilient nature of the underlying system as DNN models are capable of learning even in the presence of small amounts of lost updates. Further, these updates can be effectively recovered if needed by retraining the model on the appropriate input data. This delayed persistence allows for compressed writes to durable storage as many updates can be folded into a single parameter update, due to the additive nature of updates, between rounds of flushes. This allows update cycles to catch up to the current state of the parameter shard despite update cycles being slower.

3.3.3 Fault Tolerant Operation

There are three copies of each parameter shard in the system and these are stored on different parameter servers. The shard version that is designated as the primary is actively served while the two other copies are designated as secondary for fault tolerance. The parameter servers are controlled by a set of parameter server (PS) controller machines that form a Paxos cluster. The controller maintains in its replicated state the configuration of parameter server cluster that contains the mapping of shards and roles to parameter servers. The clients (model training machines) contact the controller to determine request routing for parameter shards. The PS controller hands out bucket

In computer science, a heartbeat is a periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a computer system.[1] Usually a heartbeat is sent between machines at a regular interval in the order of seconds. If the endpoint does not receive a heartbeat for a time—usually a few heartbeat intervals—the machine that should have sent the heartbeat is assumed to have failed.

assignments (primary role via a lease, secondary roles with primary lease information) to parameter servers and persists the lease information in its replicated state. The controller also receives heart beats from parameter server machines and relocates buckets from failed machines evenly to other active machines. This includes assigning new leases for buckets where the failed machine was the primary.

The parameter server machine that is the primary for a bucket accepts requests for parameter updates for all chunks in that bucket. The primary machine replicates changes to shards within a bucket to all secondary machines via a 2 phase commit protocol. Each secondary checks the lease information of the bucket for a replicated request initiated by primary before committing. Each parameter server machine sends heart beats to the appropriate secondary machines for all buckets for which it has been designated as primary. Parameter servers that are secondary for a bucket initiate a role change proposal to be a primary along with previous primary lease information to the controller in the event of prolonged absence of heart beats from the current primary. The controller will elect one of the secondary machines to be the new primary, assigns a new lease for the bucket and propagates this information to all parameter server nodes involved for the bucket. Within a parameter server node, the on disk storage for a bucket is modelled as a log structured block store to optimize disk bandwidth for the write heavy work load.

We have used Adam extensively over the past two years to run several training experiments. Machines did fail during these runs and all of these fault tolerance mechanisms were exercised at some point.

3.3.4 Communication Isolation

Parameter server machines have two 10Gb NICs. Since parameter update processing from a client (training) perspective is decoupled from persistence, the 2 paths are isolated into their own NICs to maximize network bandwidth and minimize interference as shown in Figure 7. In addition, we isolate administrative traffic from the controller to the 1Gb NIC.

4. EVALUATION

4.1 Visual Object Recognition Tasks

We evaluate Adam using two popular benchmarks for image recognition tasks. MNIST is a digit classification task where the input data is composed of 28x28 images of the 10 handwritten digits [20]. This is a very small benchmark with 60,000 training images and 10,000 test images that we use to characterize the baseline system performance and accuracy of trained models. ImageNet is a large dataset that contains over 15 million labeled high-resolution images belonging to around 22,000 different categories [8]. The images were gathered from a variety of sources on the web and labeled by humans using Mechanical Turk. ImageNet contains images with variable resolution but like others we down-sampled all images to a fixed 256x256 resolution and used half of the data set for training and the other half for testing. This is the largest publicly available image classification benchmark and the task of correctly classifying an image among 22,000 categories is extremely hard (for e.g., distinguishing between an American and English foxhound). Performance on this task is measured in terms of top-1 accuracy, which compares the model's top choice with the image label and assigns a score of 1 for a correct answer and 0 for an incorrect answer. No partial credit is awarded. Random guessing will result in a top-1 accuracy of only around 0.0045%. Based on our experience with this benchmark it is unlikely that human performance exceeds 20% accuracy as this task requires correctly distinguishing between hundreds of breeds of dogs, butterflies, flowers, etc.[1] We use this benchmark to characterize Adam's performance and scaling, and the accuracy of trained models.

4.2 System Hardware

Adam is currently comprised of a cluster of 120 identical machines organized as three equally sized racks connected by IBM G8264 switches. Each machine is a HP Proliant server with dual Intel Xeon E5-2450L processors for a total of 16 cores running at 1.8Ghz with 98GB of main memory, two 10 Gb NICs and one 1 Gb NIC. All machines have four 7200 rpm HDDs. A 1TB drive hosts the operating system (Windows 2012 server) and the other three HDDs are 3TB each and are configured as a RAID array. This set of machines can be configured slightly differently based on the experiment but model training machines are selected from a pool of 90 machines, parameter servers from a pool of 20 machines and image servers from a pool of 10 machines. These pools include standby machines for fault tolerance in case of machine failure.

4.3 Baseline Performance and Accuracy

We first evaluate Adam's baseline performance by focusing on single model training and parameter server machines. In addition, we evaluate baseline training accuracy by training a small model on the MNIST digit classification task.

[1] We invite people to test their performance on this benchmark available at http://www.image-net.org

4.3.1 Model Training System

We train a small MNIST model comprising around 2.5 million connections (described later) to convergence on a single model training machine with no parameter server and vary the number of processor cores used for training. We measure the average training speed computed as billions of connections trained per second (*Model connections*Training examples*Number of Epochs)/(Wall clock time*) and plot this against the number of processor cores used for training. The results are shown in Figure 8. Adam shows excellent scaling as we increase the number of cores since we allow parameters to be updated without locking. The scaling is super-linear up to 4 cores due to caching effects and linear afterwards.

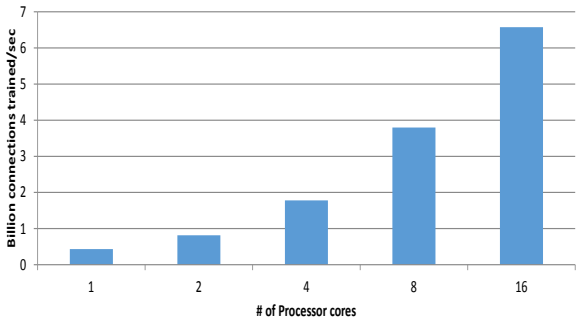

Figure 8. Model Training Node Performance.

4.3.2 Parameter Server

To evaluate the multi-core scaling of a single parameter server we collected parameter update traffic from ImageNet 22K model training runs, as MNIST parameter updates are too small to stress the system, and ran a series of simulated tests. For all tests we compare the parameter update rate that the machine is able to sustain as we increase the amount of server cores available for processing. Recall that we support two update APIs—one where the parameter server directly receives weight updates and the other where it receives activation and error gradient vectors that it must multiply to compute the weight updates. The results are shown in Figure 9. The network bandwidth is the limiting factor when weight updates are sent over the network resulting in poor performance and scaling. With a hypothetical fast network we see scaling up to 8 cores after which we hit the memory bandwidth bottleneck. When the weight updates are computed locally we see good scaling as we have tiled the computation to efficiently use the processor cache avoiding the memory bandwidth bottleneck. While our current networking technology limits our update throughput, we are still able to sustain a very high update rate of over 13 Bn updates/sec.

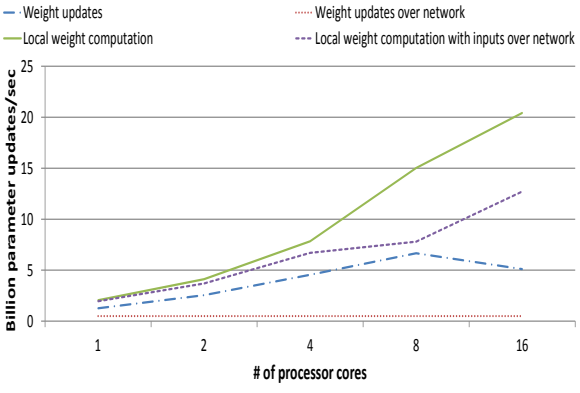

Figure 9. Parameter Server Node Performance.

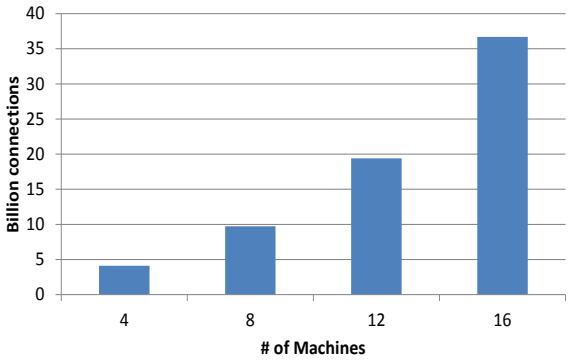

Figure 10. Scaling Model Size with more Workers.

4.3.3 Trained Model Accuracy

The MNIST benchmark is primarily evaluated in two forms. One variant transforms the training data via affine transformations or elastic distortions to effectively expand the limited training data to a much larger set resulting in the trained models generalizing well and achieving higher accuracy on the unseen test data [5, 26]. The traditional form allows no data transformation so all training has to proceed using only the limited 60,000 training examples. Since our goal here is to evaluate Adam's baseline performance on small models trained on little data we used the MNIST data without any transformation.

We trained a fairly standard model for this benchmark comprising 2 convolutional layers followed by two fully connected layers and a final ten class softmax output layer [26]. The convolutional layers used 5x5 kernels and each is followed by a 2x2 max-pooling layer. The first convolutional layer has 10 feature maps and the second has 20. Both fully connected layers use 400 hidden units. The resulting model is small and has around 2.5 million connections. The prediction accuracy results are shown in Table 1. We were

Softmax layer is typically the final output layer in a neural network that performs multi-class classification (for example: object recognition).

The name comes from the softmax function that takes as input a number of scores values (zk,k=1...Kzk,k=1...K), and squashes them into values in the range between 0, and 1 whose sum is 1. Therefore, they represent a true probability distribution.

targeting competitive performance with the state-of-the-art accuracy on this benchmark from Goodfellow et al. that uses sophisticated training techniques that we have not implemented [12]. To our surprise, we exceeded their accuracy by 0.08%. To put this improvement in perspective, it took four years of advances in deep learning to improve accuracy on this task by 0.08% to its present value. We believe that our accuracy improvement arises from the asynchrony in Adam which adds a form of stochastic noise while training that helps the models generalize better when presented with unseen data. In addition, it is possible that the asynchrony helps the model escape from unstable local minima to potentially find a better local minimum. To validate this hypothesis, we trained the same model on the MNIST data using only a single thread to ensure synchronous training. We trained the model to convergence, which took significantly longer. The result from our best synchronous variant is shown in Table 1 and indicates that asynchrony contributes to improving model accuracy by 0.24%, which is a significant increase for this task. This result contradicts conventional established wisdom in the field that holds that asynchrony lowers model prediction accuracy and must be controlled as far as possible.

Table 1. MNIST Top-1 Accuracy

Systems	MNIST Top-1 Accuracy
Goodfellow et al [12]	99.55%
Adam	**99.63%**
Adam (synchronous)	99.39%

4.4 System Scaling and Accuracy

We evaluate our system performance and scalability across multiple dimensions and evaluate its ability to train large DNNs for the ImageNet 22K classification task.

4.4.1 Scaling with Model Workers

We evaluate the ability of Adam to train very large models by partitioning them across multiple machines. We use a single training epoch of the ImageNet benchmark to determine the maximum size model we can efficiently train on a given multi-machine configuration. We do this by increasing the model size via an increase in the number of feature maps in convolutional layers and training the model for an epoch until we observe a decrease in training speed. For this test we use only a single model replica with no parameter server. The results are shown in Fig. 10 and indicate that Adam is capable of training extremely large models using a relatively small number of machines. Our 16 machine configuration is capable of training a 36 Bn connection model. More importantly, the size of models we can train efficiently increases super-linearly as we partition the model across more machines. Our measurements indicate that this is due to cache effects where larger portions of the working sets of model layers fits in the L3 cache as the number of machines is increased. While the ImageNet data set does not have sufficient training data to train such large models to convergence these results indicate that Adam is capable of training very large models with good scaling.

4.4.2 Scaling with Model Replicas

We evaluate the impact of adding more model replicas to Adam. Each replica contains 4 machines with the ImageNet model (described later) partitioned across these machines. The results are shown in Figure 11 where we evaluated configurations comprising 4, 10, 12, 16, and 22 replicas. All experiments used the same parameter server configuration comprised of 20 machines. The results indicate that Adam scales well with additional replicas. Note that the configuration without a parameter server is merely intended as a reference for comparison since the models cannot jointly learn without a shared parameter server. While the parameter server does add some overhead the system still exhibits good scaling.

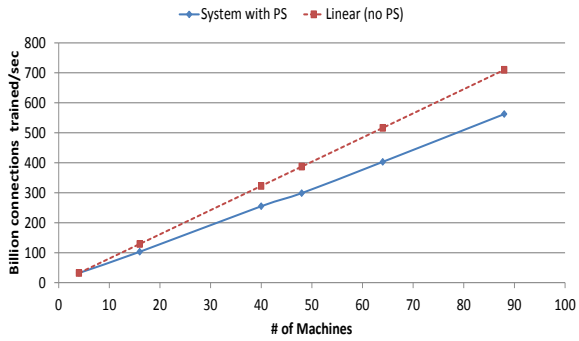

Figure 11. System scaling with more Replicas.

4.4.3 Trained Model Accuracy

We trained a large and deep convolutional network for the ImageNet 22K category object classification task with a similar architecture to those described in prior work [17, 27]. The network has five convolutional layers followed by three fully connected layers with a

Table 2. ImageNet 22K Top-1 Accuracy.

Systems	ImageNet 22K Top-1 Accuracy
Le et al. [18]	13.6%
Le et al. (with pre-training) [18]	15.8%
Adam	**29.8%**

final 22,000-way softmax. The convolutional kernels range in size from 3x3 to 7x7 and the convolutional feature map sizes range from 120 to 600. The first, second and fifth convolutional layers are followed by a 3x3 max-pooling layer. The fully-connected layers contain 3000 hidden units. The resulting model is fairly large and contains over 2Bn connections. While Adam is capable of training much larger models the amount of ImageNet training data is a limiting factor in these experiments.

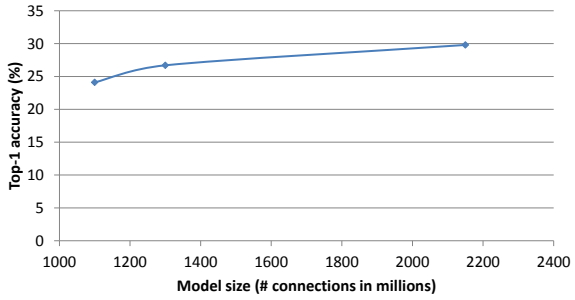

Figure 12. Model accuracy with larger models.

We trained this model to convergence in ten days using 4 image servers, 48 model training machines configured as 16 model replicas containing 4 machines per replica and 10 parameter servers for a total of 62 machines. The results are shown in Table 2. Le et al. held the previous best top-1 accuracy result on this benchmark of 13.6% that was obtained by training a 1Bn connection model on 2,000 machines for a week (our model exceeds 13.6% accuracy with a single day of training using 62 machines). When they supplemented the ImageNet training data with 10 million unlabeled images sampled from Youtube videos, which they trained on using 1,000 machines for 3 days, they were able to increase prediction accuracy to 15.8%. Our model is able to achieve a new world record prediction accuracy of 29.8% using only ImageNet training data, which is a dramatic 2x improvement over the prior best.

To better understand the reasons for this accuracy improvement, we used Adam to train a couple of smaller models to convergence for this task. The results are shown in Figure 12 and indicate that training larger models increases task accuracy. This highlights the importance of Adam's efficiency and scalability as it enables training larger models. In addition, our 1.1 Bn connection model achieves 24% accuracy on this task as compared to prior work that achieved 13.6% accuracy with a similar size model. While we are unable to isolate the impact of asynchrony for this task as the synchronous execution is much too slow, this result in conjunction with the MNIST accuracy data provides evidence that asynchrony contributes to the accuracy improvements. The graph also appears to suggest that improvements in accuracy slow down as the model size increases but we note that the larger models are being trained with the same amount of data. It is likely that larger models for complex tasks require more training data to effectively use their capacity.

4.4.4 Discussion

Adam achieves high multi-threaded scalability on a single machine by permitting threads to update local parameter weights without locks. It achieves good multi-machine scalability through minimizing communication traffic by performing the weight update computation on the parameter server machines and performing asynchronous batched updates to parameter values that take advantage of these updates being associative and commutative. Finally, Adam enables training models to high accuracy by exploiting its efficiency to train very large models and leveraging asynchrony to further improve accuracy.

5. RELATED WORK

Due to the computational requirements of deep learning, deep models are popularly trained on GPUs [5, 14, 17, 24, 27]. While this works well when the model fits within 2-4 GPU cards attached to a single server, it limits the size of models that can be trained. Consequently the models trained on these systems are typically evaluated on the much smaller ImageNet 1,000 category classification task [17, 27].

Recent work attempted to use a distributed cluster of 16 GPU servers connected with Infiniband to train large DNNs partitioned across the servers on image classification tasks [6]. Training large models to high accuracy typically requires iterating over vast amount of data. This is not viable in a reasonable amount of time unless the system also supports data parallelism. Unfortunately the mismatch in speed between GPU compute and network interconnects makes it extremely difficult to support data parallelism via a parameter server. Either the GPU must constantly stall while waiting for model parameter updates or the models will likely diverge due to insufficient synchronization. This work did not support data parallelism and the large models trained had lower accuracy than much smaller models.

The only comparable system that we are aware of for training large-scale DNNs that supports both model and data parallelism is the DistBelief system [7]. The system has been used to train a large DNN (1 billion connections) to high accuracy on the ImageNet 22K classification task but at a significant compute cost of using 2,000 machines for a week. In addition, the

system exhibits poor scaling efficiency and is not a viable cost-effective solution.

GraphLab [21] and similar large scale graph processing frameworks are designed for operating on general unstructured graphs and are unlikely to offer competitive performance and scalability as they do not exploit deep network structure and training efficiencies.

The vision and computer architecture community has started to explore hardware acceleration for neural network models for vision [3, 4, 10, 16, 22]. Currently, the work has concentrated on efficient feed-forward evaluation of already trained networks and complements our work that focuses on training large DNNs.

6. CONCLUSIONS

We show that large-scale commodity distributed systems can be used to efficiently train very large DNNs to world-record accuracy on hard vision tasks using current training algorithms by using Adam to train a large DNN model that achieves world-record classification performance on the ImageNet 22K category task. While we have implemented and evaluated Adam using a 120 machine cluster, the scaling results indicate that much larger systems can likely be effectively utilized for training large DNNs.

7. ACKNOWLEDGMENTS

We would like to thank Patrice Simard for sharing his gradient descent toolkit code that we started with as a single machine reference implementation. Leon Bottou provided valuable guidance and advice on scalable training algorithms. John Platt served as our machine learning consultant throughout this effort and constantly shared valuable input. Yi-Min Wang was an early and constant supporter of this work and provided the initial seed funding. Peter Lee and Eric Horvitz provided additional support and funding. Jim Larus, Eric Rudder and Qi Lu encouraged this work. We would also like to acknowledge the contributions of Olatunji Ruwase, Abhishek Sharma, and Xinying Song to the system. We benefitted from several discussions with Larry Zitnick, Piotr Dollar, Istvan Cseri, Slavik Krassovsky, and Sven Groot. Finally, we would like to thank our reviewers and our paper shepherd, Geoff Voelker, for their detailed and thoughtful comments.

8. REFERENCES

[1] Bengio, Y., and LeCun, Y. 2007. Scaling Learning Algorithms towards AI. In *Large-Scale Kernel Machines*, Bottou, L. et al. (Eds), MIT Press.

[2] Bottou, L., 1991. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nîmes 91*, EC2, Nimes, France.

[3] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *International symposium on Computer Architecture*, ISCA'10.

[4] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS'14.

[5] Ciresan, D. C, Meier, U., and Schmidhuber, J. 2012. Multicolumn deep neural networks for image classification. In *Computer Vision and Pattern Recognition*. CVPR'12.

[6] Coates, A., Huval, B., Wang, T., Wu, D., Ng, A., and Catanzaro, B. 2013. Deep Learning with COTS HPC. In *International Conference on Machine Learning*. ICML'13.

[7] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q., and Ng, A. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*. NIPS'12.

[8] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition*. CVPR '09.

[9] Faerber, P., and Asanović, K. 1997. Parallel neural network training on Multi-Spert. In *IEEE 3rd International Conference on Algorithms and Architectures for Parallel Processing* (Melbourne, Australia, December 1997).

[10] Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., and LeCun, Y. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition* Workshop (June 2011), pages 109–116.

[11] Fukushima, K. 1980. Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. In *Biological Cybernetics*, 36(4): 93-202.

[12] Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. 2013. Maxout

Networks. In *International Conference on Machine Learning*. ICML'13.

[13] Hahnloser, R. 2003. Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks. In *Neural Computing*. (Mar. 2003), 15(3):621-38.

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

[15] Hubel, D. and Wiesel, T. 1968. Receptive fields and functional architecture of monkey striate cortex. In *Journal of Physiology* (London), 195, 215–243.

[16] Kim, J., Member, S., Kim, M., Lee, S., Oh, J., Kim, K. and Yoo, H. 2010. A 201.4 GOPS 496 mW Real-Time Multi-Object Recognition Processor with Bio-Inspired Neural Perception Engine. In *IEEE Journal of Solid-State Circuits*, (Jan. 2010), 45(1):32–45.

[17] Krizhevsky, A., Sutskever, I., and Hinton, G. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. NIPS'12.

[18] Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J., and Ng, A. 2012. Building high-level features using large scale unsupervised learning. In *International Conference on Machine Learning*. ICML'12.

[19] LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. In *Neural Computation*, 1(4):541-551.

[20] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 86(11):2278–2324, (Nov. 1998).

[21] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. 2012. Distributed GraphLab: A framework for machine learning in the cloud. In *International Conference on Very Large Databases.* VLDB'12.

[22] Maashri, A., Debole, M., Cotter, M., Chandramoorthy, N., Xiao, Y., Narayanan, V., and Chakrabarti, C. 2012. Accelerating neuromorphic vision algorithms for recognition. In *Proceedings of the 49th Annual Design Automation Conference*, DAC'12.

[23] Niu, F., Retcht, B., Re, C., and Wright, S. 2011. Hogwild! A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*. NIPS'11.

[24] Raina, R., Madhavan, A., and Ng., A. 2009. Large-scale deep unsupervised learning using graphics processors. In *International Conference on Machine Learning.* ICML'09.

[25] Rumelhart, D., Hinton, G., and Williams, R. 1986. Learning representations by back-propagating errors. In *Nature* 323 (6088): 533–536.

[26] Simard, P., Steinkraus, D., and Platt, J. 2003. Best Practices for Convolutional Neural Networks applied to Visual Document Analysis. In *ICDAR*, vol. 3, pp. 958-962.

[27] Zeiler, M. and Fergus, R. 2013. Visualizing and Understanding Convolutional Networks. In *Arxiv 1311.2901*. http://arxiv.org/abs/1311.2901