Parallel Training of Fully-connected Deep Neural Network Using Layer-wise Model Partitioning

Lin Wang[a,*], Xingfu Wang[a], Yan Xiong[a]

[a] *University of Science and Technology of China, Hefei 230027, P.R. China.*

Abstract

The large model size and massive training examples make deep neural network (DNN) a powerful model for classification. However, these two factors also slow down the training procedure. Currently the most widely adopted acceleration of DNN training, asynchronous stochastic gradient descent (ASGD), are also facing increasingly noticeable bottleneck of scalability as the training data size continue to expand. In this paper we propose another parallel scheme to speedup stochastic gradient descent (SGD) training of DNN on multi-GPU, which fits more inherently to the deep nature of modern neural architectures and achieves better performance in terms of parallel efficiency as compared with ASGD. Experimental results show that it achieves 3.3 and 6.8 times end-to-end speed-up using 4 and 8 GPUs than a single one, at parallelization efficiency of 0.83 and 0.85, respectively, at no loss of recognition accuracy.

Keywords: deep neural network, speech recognition, asynchronous SGD, GPU parallelization, model partitioning

1. Introduction

While convolutional deep network gains great success for image classification[1, 2, 3, 4, 5], DNNs of fully-connected type has also shown its effectiveness in a variety of machine learning tasks, especially in speech recognition[6, 7, 8, 9].

*Corresponding author

Email address: xiaquhet@mail.ustc.edu.cn (Lin Wang)

In fact, DNN has already replaced conventional Gaussian Mixture Models (GMM) to become the state-of-the-art pattern classifier in speech recognition systems[10, 11, 12]. However, we have to pay immense computation cost correspondingly to train them.

Many attempts to parallelize the training of deep networks rely on asynchronous, lock-free optimization (asynchronous stochastic gradient descent, ASGD) [13, 14, 15, 16, 17, 18, 19]. ASGD allows each GPU to transmit and calculate gradient asynchronously on their own schedule. Although it has been reported to be able outperform synchronous data splitting solutions [17, 18, 19], there is still a lack of detailed analysis to quantitatively verify their parallel effectiveness.

Moreover, while DNNs for vision are commonly convolutional with local connectivity[20, 21, 22], typical DNNs for general classifications are fully connected between layers[23, 24]. This global connectivity makes parallelizing back-propagation over multiple compute nodes inefficient. E.g., Google's DistBelief system successfully utilizes 16,000 cores for the ImageNet task through asynchronous SGD[21], while for a fully-connected DNN with 42M parameters, a 1,600-core DistBelief[13] is only marginally faster than a single recent GPGPU. Therefore, how to utilize multiple GPUs efficiently on one DNN model is a critical issue to speed up the training process.

In this paper we make both empirical and theoretical analyses of ASGD to reveal its underlying scalability bottleneck, and propose another parallel design to speed up DNN training on multi-GPU platforms, which is able to overcome the bandwidth problem of ASGD in a more fundamental way.

The outline of this paper is as follows: Section 2 discusses previous works related to DNN training acceleration via parallel computing. Section 3 introduces the background of DNN used in speech recognition and the paradigm of ASGD. Section 4 theoretically evaluates the parallel computation of ASGD, by establishing analytical expressions, using measured hardware parameters such as computation speed and data-exchange bandwidth. Section 5 describes our parallel design that partitions the computation with respect to layers, which is

the main technical contribution of this paper. Section 6 verifies the effectiveness of our approach and presents experimental results. Section 7 concludes.

2. Related Work

Scaling up deep learning algorithms has been shown to lead to increased performance on benchmark tasks[1, 6, 7, 13, 25], hence parallelization of DNN training has been a popular topic since the revival of neural networks.

Dean et al. build DistBelief[13], a distributed system capable of training deep networks on thousands of machines using stochastic and batch optimization procedures. In particular, they highlight asynchronous SGD and batch L-BFGS. Distbelief exploits both data parallelism and model parallelism.

Chilimbi et al. [14] build Project Adam, a system for training deep networks on hundreds of machines using asynchronous SGD. Iandola et al. build FireCaffe[26], a data-parallel system that achieves impressive scaling using naive parallelization in the high-performance computing setting. They minimize communication overhead by using a tree reduce for aggregating gradients in a supercomputer with Cray Gemini interconnects.

More recently, distributed model averaging is used in [27], where separate models are trained on multiple nodes using different partitions of data, and model parameters are averaged after each epoch. DeepSpark[19] proposes a framework for training deep networks in Spark to facilitate model deployment with existing distributed data-processing pipelines. However, the widely-popular batch-processing frameworks like Spark were not designed to support the asynchronous and communication-intensive workloads of distributed deep learning systems, the performance obtained are inferior compared with those customized implementations.

These existing systems are mostly data-parallel or ASGD based. As will be discussed in Section 4, ASGD theoretically faces the problem of a saturation point, caused by its intensive communication requirement and memory inefficiency.

A separate line of work to speedup DNN training uses a model-parallel system[22], where each GPU is responsible for only a piece of the whole network, reduces bandwidth requirements but still requires frequent synchronization (usually once for each forward or backward-propagation step).

Our approach is also based on the idea of model parallelism. Rather than "striping" the parameter matrix, we instead choose to distribute the layers themselves across GPUs, in order to better fit to fully-connected structures. Our purpose is to exploit the parallelism formed by successive layers when features vectors are flowing forwards and backwards through the model. Experimental results confirm that it is able to further reduce bandwidth and synchronization requirements compared to existing methods.

3. Theoretical Background

This section recaps the structure of DNN used in more general pattern recognition applications, such as the observed feature vectors of a speech recognition system. An insight into the minibatch size is also given in this section to put forward the SGD strategy in BP training.

3.1. Fully connected DNN structure

Figure 1 shows the structure of fully-connected DNN, typically used in speech recognition systems, which is also known as multi-layer perceptron (MLP) [28].

The term "deep" means the multiple hidden layers in the network. As commonly used in speech recognition systems, the number of hidden layers is often 5, 7 or more. Each hidden layer consists of hidden units with a few thousands, e.g. 2048, and the output layer with much more, e.g. 10k.

Consider a model with $(L-1)$ hidden layers, N_i units for input layer, N_o for output layer, and N_h for hidden layers, then the total weight parameters number is $N_W = N_h \cdot \big(N_i + (L-2) \cdot N_h + N_o\big)$. Let $L-1 = 7, N_i = 400, N_h = 2048, N_o = 10\,k$ (even larger in practice usually), N_W tends to be in the order of 10^8, which is a root cause for parallelization problem.

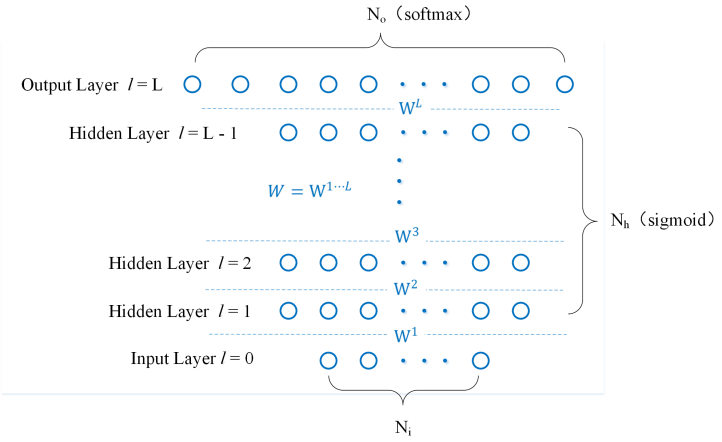


Figure 1: deep neural network with dimensions $N = \{L, N_i, N_h, N_o\}$

The forward propagation in the network can be viewed as the probability calculation in speech recognition, and the training is done by the well-known back-propagation (BP) algorithm [29]. BP relies on iterative gradient descent optimizations to minimize the network errors.

3.2. Minibatch based stochastic gradient descent

Gradient descent finds a local minimum by taking steps proportional to the negative gradient of the function at current point, so that:

$$W(\lambda + 1) = W(\lambda) - \eta \nabla E\big(W(\lambda)\big) \tag{1}$$

where λ is the model version, η is the learning rate, W is the model parameter and E is the error function.

the gradient ∇E can be calculated after all the training patterns have been processed (epoch mode BP). The speed of convergence and the accuracy obtained using epoch mode BP are usually inferior compared with *stochastic gradient descent (SGD)*[30], where the error function gradient ∇E_n is calculated

using batch of data sampled randomly instead of the whole dataset:

$$E(W) = \sum_{k=1}^{K} E_k(W)$$
$$W(\lambda + 1) = W(\lambda) - \eta \nabla E_k\big(W(\lambda)\big) \tag{2}$$

where K is the number of mini batches.

3.3. Asynchronous SGD strategy

The skeleton of ASGD can be summerized as Algorithm 1. In the beginning, the model is initialized and stored in parameter server. Each GPU works as a client, and performs the loop described in Figure 2 repeatedly until all training data is processed:

Step 1. A worker node requests a minibatch from training set which is guaranteed by the mutex locking protocol.

Step 2. GPU gets the current model $\big($e.g. model $W(\lambda)\big)$ from server's memory (If the current model is being updated, then waits and retries).

Step 3. Having received the data and model, GPU calculates the gradient based on minibatch as described in Section 3.2.

Step 4. Subsequently it transmits the gradient back to the server to update the current model (not model $W(\lambda)$ usually, because it may be updated by other clients already).

Step 5. The model updating procedure is executed by server.

Algorithm 1: Asynchronous Stochastic Gradient Descent

```
Require: Mutex, LoadModel(), UpdateModel()
W_server ← LoadModel()
loop
    Mutex.Lock()
    X ← get next mini-batch
    W_client ← download W_server
```

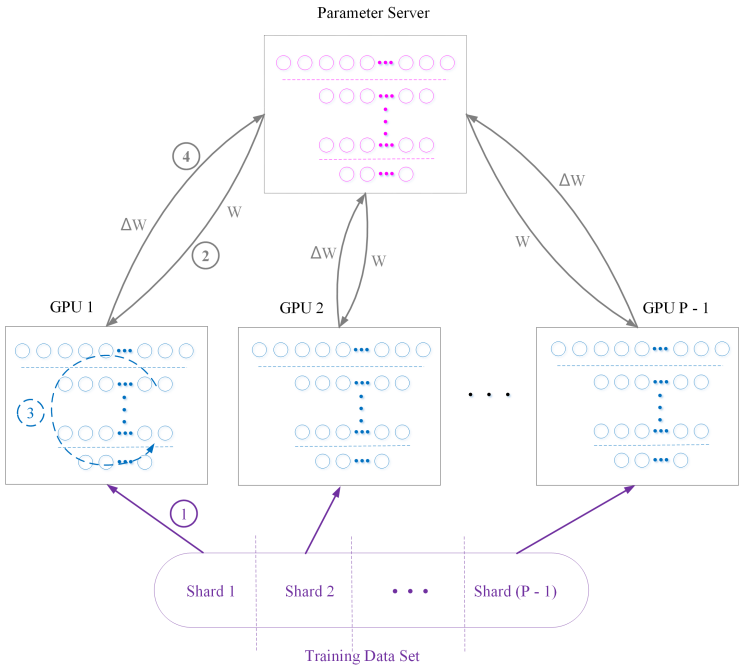


Figure 2: ASGD applied on multi-GPU cards in a server

```
      Mutex.Unlock()
      if X is empty then
         exit
      end if
      g_client ← calculate gradient based on X and W_client
      g_server ← upload g_client
      Mutex.Lock()
      W_server ← UpdateModel(W_server, g_server)
      Mutex.Unlock()
   end loop
```

3.4. A better understanding of mini batch size

If we write SGD as an iteration over individual samples $o(t)$ (indexed by sample index t) rather than, as common, over minibatches:

$$W(t+1) = W(t) - \eta \nabla E_W\big(o(t)\big)|_{W=W(\tau)} \tag{3}$$

Here, $W(t)$ denotes the model at "current" sample index t, while $W(\tau)$ is meant to denote a slightly "outdated" model at index $\tau \leq t$. It is this model that the

partial gradient of the objective function $E_W\big(o(t)\big)$ for the current sample vector $o(t)$ is evaluated on.

With this, mini-batching can be described by defining $\tau = t-(t \mod b)$ with mini-batch size b. I.e., τ is rounded down to multiples of b. As long as t falls within the same mini-batch, the formula simply sums up individual framesą́ gradients computed on the same model $W(\tau)$.

This notation is useful because more generally, $\tau < t$ means that gradients are computed using a model that is $(t - \tau)$ samples "outdated". Any optimal variant of data parallelism necessarily implies some form of such delayed update — new samples are processed while results from previous samples are still being transferred concurrently. A popular variant of this is asynchronous SGD, or ASGD [13], where τ varies non-deterministically across model parameters. Hence, Eq.(3) allows us to understand more complex forms of delayed updates as something qualitatively similar to mini-batching.

Figure 3 shows training times and corresponding phone error rates on the TIMIT corpus[31] with the same 1024 hidden units but different hidden layers and mini batch sizes. TIMIT is a small vocabulary benchmark[1] widely used for speech recognition.

Experiments show that the size of mini batch influences both the training time and performance significantly. Undersized mini batch (extremely, down to $b = 1$) will slow down the training due to the poor utilization of GPU computation units, while oversized mini batch will degrade the performance. Moreover, training tends to diverge with oversized mini batch (error rates increase obviously for size $b = 2048$ and $b = 4096$ in Figure 3).

In fact, increasing mini batch size means calculating more frames at the same time, which will speed up training theoretically, and it is true w.r.t. small mini batch size. But for larger size, it contributes little to reduce number of model updates but degrades performance significantly.

The above analysis tell us that minibatch size b has to be retained in a proper

[1] `https://catalog.ldc.upenn.edu/LDC93S1`

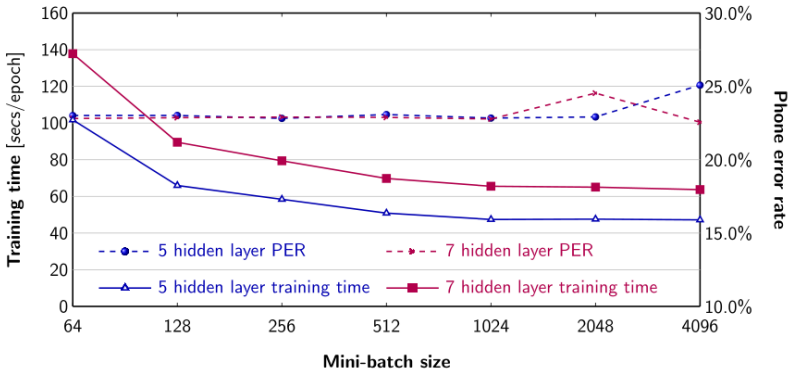


Figure 3: Training times and phone error rates according to different mini batch sizes and numbers of hidden layers on the TIMIT corpus

range and increasing minibatch size directly doesnąŕt help in speeding up the training. In our trainings, we limit it to 256 for the first 10h of data to ensure convergence, and then relax it to a standard size of $b = 1024$ in later epochs.

4. Performance Analysis of ASGD

In this section, we introduce a formalism to estimate the efficiency of ASGD parallel computation. The speedup S and efficiency E of a parallel algorithm can be given by the ratio:

$$\begin{aligned} S &= t_{ser}/t_{par} \\ E &= S/P \end{aligned} \tag{4}$$

respectively, where t_{ser} and t_{par} are the sequential and the parallel computation times, P is the number of compute nodes.

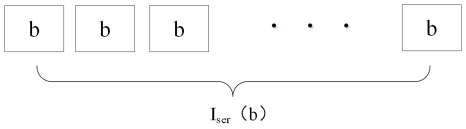
Let $I_{ser}(b)$ and be the number of serial iterations of SGD required to obtain an accuracy of a when training with a batch size of b (when we say accuracy, we are referring to test accuracy). Suppose that computing the gradient over a batch of size b requires $T_{calc}(b)$ units of time. Then the running time required

to achieve an accuracy of a with serial training is

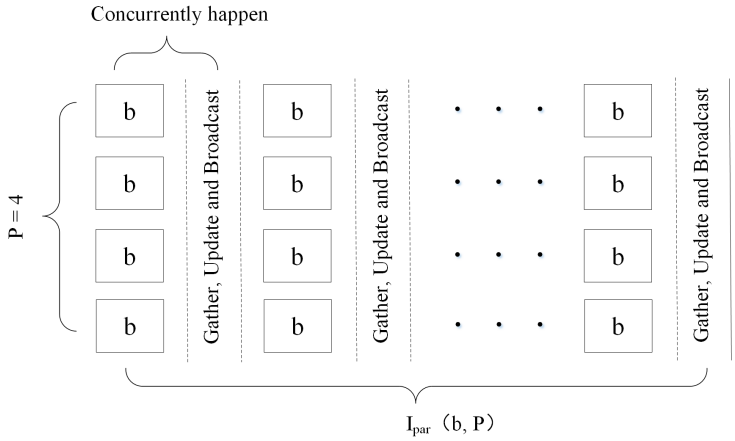
$$t_{ser} = I_{ser}(b) \cdot T_{calc}(b) \tag{5}$$

4.1. Speedup of synchronized data parallel SGD

To begin with, we consider the theoretical speedup of the synchronized data parallelism, where each GPU keeps a complete copy of the neural network parameters but computes a gradient using a minibatch from different subset of the training data (Figure 4).



(a) A serial run of SGD. Each block corresponds to a single SGD update with batch size b. $I_{ser}(b)$ is the number of iterations required to achieve a predefined accuracy a.



(b) A parallel run of SGD on $P = 4$ machines under synchronized data parallel. During the time when each worker runs SGD with batch size b for 1 iteration $\left(T_{calc}(b)\right)$, all workers need to send their model gradients back to the host, the host averages them and redistributes the updated result to the workers $\left(T_{comm} \cdot 2P\right)$.

Figure 4: Computational models for serial and synchronous data parallel SGD schemes

We use $I_{par}(b, P)$ to denote the number of parallel iterations required to achieve the same accuracy a on P machines under synchronized data parallel

scheme, T_{comm} to denote the time per mini-batch per-node for concurrent one-way inter-node communication, provided that computation and data exchange happen concurrently with overlap, a synchronized data parallelization could in principle achieve the same accuracy a in time:

$$t_{par} = \left(\frac{I_{par}(b,P)}{P}\right) \cdot \max\left(T_{calc}(b), T_{comm} \cdot 2P\right) \tag{6}$$

As discussed in Section 3.4, we can expect similar convergence behavior as long as the update delay $(t-\tau) = (t \mod b)$ stays in a similar range. Thus

$$I_{par}(b,P) \approx I_{ser}(b) \tag{7}$$

Taking Eq.(5),(7) and (6) into Eq.(4), we have:

$$\begin{aligned} S &= t_{ser}/t_{par} = P \cdot \frac{T_{calc}(b)}{\max\left(T_{calc}(b), T_{comm} \cdot 2P\right)} \\ E &= S/P = \frac{T_{calc}(b)}{\max\left(T_{calc}(b), T_{comm} \cdot 2P\right)} \end{aligned} \tag{8}$$

This result (Eq.(8)) tell us that the condition on which the parallel system reaches its optimality is that when communication and processing time are balanced:

$$T_{calc}(b) = T_{comm} \cdot 2P \tag{9}$$

that is, simultaneously saturating the communication channel and the processing resources:

- If the communication channel is not saturated $\left(T_{calc}(b) > T_{comm} \cdot 2P\right)$, then the system could be improved by parallelizing more.
- If the processing resources are not saturated $\left(T_{calc}(b) < T_{comm} \cdot 2P\right)$, it could be improved by parallelizing (communicating) less or by processing more data.

If Eq.(9) holds for Eq.(8), we can have a perfect $S = P$ and $E = 100\%$, respectively, however, we can see from empirical estimates later in Section 4.3

and 4.4, even with high-speed PCI bus within a single server, this is far from reality.

4.2. Notations

For the convenience of readers' reference, here we list the notations used for performance analysis in this paper :

$\Longrightarrow$ Basic symbols:

N	$N = \{L, N_i, N_h, N_o\}$, network dimensions (Figure.1).
l	Index of the network layer, $(l = 0, 1, \cdots, L)$.
i	Index of the neuron in a given layer l, $i = 0, 1, \cdots, N^l - 1$.
t	Index of the training example, i.e. a $10\,ms$ signal frame sampled at each time step in speech recognition, $t = 0, 1, \cdots, T - 1$.
k	Index of the minibatch presented as the input of the neural network, $k = 1, \cdots, K(K = T/b)$.
b	Size of minibatch.
P	Number of compute nodes (i.e. GPU cards in our hardware configuration).

$\Longrightarrow$ For model description and footprint requirement:

W^l	$W^l = \{\omega_{ij}^l\}$, the synaptic matrix containing all the coefficients ω_{ij}^l whose presynaptic neuron i belongs to layer $l-1$, postsynaptic neuron j belongs to layer l.
$W^{l_1 \cdots l_2}$	The synaptic matrixes related to the layers l_1 to l_2 of W^l.
W	$W = W^{1 \cdots L}$, the synaptic matrix of the network.
N_W	$N_W = N_h \cdot \big(N_i + (L-1) \cdot N_h + N_o\big)$, number of weights of W.
ω_B	Byte size of ω, usually $\omega_B = 4$ for single-precision floating point.
M	$M = N_W \cdot \omega_B$, the amount of memory required to store W, also the amount of data to transmit during T_{comm}.

$\Longrightarrow$ For algorithm description:

O $O = \{o(t) = (x(t), d(t))\}$, the training set containing T examples, each $o(t)$ consists of feature vector $x(t)$ with its corresponding desired output target $d(t)$.

$V^l(t)$ $V = \{\nu_i\}$, input vector of the neurons of layer l for example t.

$\sigma^l(t)$ $\sigma = \{\sigma_i\}$, state vector of the neurons of layer l for example t:

$$\sigma_i = \begin{cases} sigmoid(\nu_i) = \dfrac{1}{1+\exp(-\nu_i)} & l = 1, \cdots, L-1 \\ & \text{F.3 Sigmoid Function} \\ softmax(\nu_i) = \dfrac{\exp(\nu_i)}{\sum_{N^l}\exp(\nu)} & l = L \\ & \text{F.2 Softmax Function} \end{cases}$$

$E(t)$ $E = \{E_i\}$, cross entropy error between the target vector d and the network output σ^L for example t:

$$E_i = -d_i \cdot \ln(\sigma_i^L) - (1-d_i) \cdot \ln(1-\sigma^L)$$

$B^l(t)$ $B = \{b_i\}$, the derivative of the error E with respect to the state σ^l for neurons of layer l for example t:

$$b_i^l = \frac{\partial E}{\partial \sigma_i^l} = \begin{cases} cross-entropy'(\sigma_i^L) = \dfrac{1-d_i}{1-\sigma_i^L} - \dfrac{d_i}{\sigma_i^L}, & l = L \\ & \text{B.0 Cross Entropy Derivation} \\ \sum\limits_{j}^{N^{(l+1)}} \left(\delta_j^{(l+1)} \cdot \dfrac{\partial \nu_j^{(l+1)}}{\partial \sigma_i^l}\right) = \sum\limits_{j}^{N^{(l+1)}} \left(\delta_j^{(l+1)} \cdot \omega_{ij}^l\right), & l = 1, \cdots, L-1 \\ & \text{B.2 Back Propagation} \end{cases}$$

$\delta^l(t)$ $\delta = \{\delta_i\}$, the derivative of the error E with respect to the input V^l for neurons of layer l for example t:

$$\delta_i^l = \frac{\partial E}{\partial \nu i^l} = b_i^l \cdot \frac{\partial \sigma_i^l}{\partial \nu_i^l} = \begin{cases} b_i^l \cdot softmax'(\nu_i^l) & l = L \\ b_i^l \cdot sigmoid'(\nu_i^l) & l = 1, \cdots, L-1 \end{cases}$$
$$= b_i^l \cdot \left[\sigma_i^l \cdot (1 - \sigma_i^l)\right], \text{ for all } \quad l$$

B.1 Activation Function Derivation

4.3. Empirical estimates of T_{calc}

T_{calc} is dominated by 3 large size matrix products — forward propagation ($V = \sigma \cdot W$, F.1), error back propagation ($B = \delta \cdot W^T$, B.2), and gradient computation ($\Delta W = \sigma^T \cdot \delta$, B.3) — and matrix extensions of 2 kinds of scalar operations — the activation functions (F.2, F.3) and their derivatives (B.1) for each elements of matrix. This can be illustrated by writing BP in matrix form as Algorithm 2:

Algorithm 2: BP in matrix representation

Require: Feature matrix $X_{(b\times N_i)}$ and target matrix $D_{(b\times N_o)}$ for minibatch k.

$\sigma^0_{(b\times N_i)} = X_{(b\times N_i)}$

for l in $range(1, L)$ do /* Feed−forward step */

  $V^l_{(b\times N^l)} = \sigma^{(l-1)}_{(b\times N^{(l-1)})} \cdot W^l_{(N^{(l-1)}\times N^l)}$ F.1 Forward Propagation

  if $l == L$

    $\sigma^l_{(b\times N^l)} = softmax\left(V^l_{(b\times N^l)}\right)$ F.2 Softmax Function

  else

    $\sigma^l_{(b\times N^l)} = sigmoid\left(V^l_{(b\times N^l)}\right)$ F.3 Sigmoid Function

end for

$B^L_{(b\times N_o)} = \left(1_{(b\times N_o)} - D_{(b\times N_o)}\right)./\left(1_{(b\times N_o)} - \sigma^L_{(b\times N_o)}\right) - \left(D_{(b\times N_o)}./\sigma^L_{(b\times N_o)}\right)$

B.0 Cross Entropy Derivation

for l in $range(L, 1)$ do /* Back−propagation step */

  $\delta^l_{(b\times N^l)} = \left(1_{(b\times N^l)} - \sigma^l_{(b\times N^l)}\right).*\sigma^l_{(b\times N^l)}.*B^l_{(b\times N^l)}$

  B.1 Activation Function Derivation

  $B^{(l-1)}_{(b\times N^{(l-1)})} = \delta^l_{(b\times N^l)} \cdot \left(W^l_{(N^{(l-1)}\times N^l)}\right)^T$ B.2 Back Propagation

  $\Delta W^l_{(N^{(l-1)}\times N^l)} = -\eta \cdot \left(\sigma^{(l-1)}_{(b\times N^{(l-1)})}\right)^T \cdot \delta^l_{(b\times N^l)} + \beta \cdot \Delta W^l_{(N^{(l-1)}\times N^l)}$

  B.3 Gradient Computation

$W^l_{(N^{(l-1)} \times N^l)} = W^l_{(N^{(l-1)} \times N^l)} + \Delta W^l_{(N^{(l-1)} \times N^l)}$ B.4 Weight Update

end for

In Algorithm 2 $./$ and $.*$ represent the element-by-element division and multiplication, respectively. T indicates transposition, the superscript is the number of the network layer and the subscript gives the dimensions of the matrices.

We implement matrix multiplications and element-wise operations with the hardware-optimized cublasSgemm function from CuBLAS library[32] and CUDA kernels[33], respectively, on an NVidia Tesla K20 GPU (Table 1). CUDA kernels are SIMD instructions executed in parallel by all the GPU stream processors. Thus, a kernel applies the element-wise operations in parallel to all the units of the matrix.

Table 1: Processing speed in batches per second (bps), the timing profile normalized as $T_{calc} = 100\%$.

Implementation		Dimension[(a)]	Order[(b)]	Time (%)
Feed-forward Step		20.6 bps[(c)]	48.5 ms/batch	**32.5**
F.1	cublasSgemm	$b \cdot N_W$	$O(10^{10})$	22.8
F.2	CUDA kernel	$b \cdot N_o$	$O(10^7)$	7.2
F.3	CUDA kernel	$b \cdot N_h \cdot (L-1)$	$O(10^7)$	2.5
Back-propagation Step		28.3 bps	35.3 ms/batch	**23.7**
B.0	CUDA kernel	$b \cdot N_o$	$O(10^7)$	1.2
B.1	CUDA kernel	$b \cdot \big(N_o + N_h \cdot (L-1)\big)$	$O(10^7)$	2.4
B.2	cublasSgemm	$b \cdot N_W$	$O(10^{10})$	20.1
Model Update Step		26.7 bps	37.5 ms/batch	**25.1**
B.3	cublasSgemm	$b \cdot N_W$	$O(10^{10})$	17.0
B.4	CUDA kernel	N_W	$O(10^7)$	8.1
Other cause of overhead[(d)]				18.7

To be continued

Table 1 (Continued)

End-to-end BP (T_{calc})	6.70 bps	149.3 ms/batch	100

[a] Dimensions of matrixes involved in cublasSegmm and CUDA kernel operations, respectively.
[b] Order of magnitude of the computational complexity for cublasSegmm and parallel granularity for kernel operations, respectively.
[c] Processing speed in batches per second (bps), for both the entire T_{calc}, and broken out by the three main steps, forward propagation, error back propagation, and model-parameter update.
[d] Included in T_{calc}, in addition to the above three main steps, such as data loading, objective-function tracking, etc.

With batch size $b = 1024$ and model size $N_W = 45\,M$ (network dimensions N set as in Section 3.1), T_{calc} is tested to be $T_{calc} \approx 149.3\,ms/batch$ on a single K20.

4.4. Empirical estimates of T_{comm}

We prepared a single-server environment with 8 K20 cards accessing each other via PCI-E Gen3 x16 bus (refer to Section 6.1 for detailed setup configurations). PCI-E 3.0 allows direct data exchange between GPU devices without CPU intervention.

For a typical 7-hidden-layer DNN in the order of 10^8 parameters, each client would require the gathering/redistribution of $M \approx 180$ MB worth of gradients and another 180 MB of model parameters, per minibatch. Duplex x16 bandwidth of K20 when exchanging chunks of 180 MB was measured at $7.6\,GB/s$ with $T_{comm} \approx 23.7\,ms/minibatch$. We get $T_{calc}(b) < T_{comm} \cdot 2P$ with a not-so-ideal speedup $S = T_{calc}(b)/\big(2 \cdot T_{comm}\big) \approx 3.1$ (Eq.(8)).

4.5. Why data parallel does not scale well

The scaling problem of data parallel approach can be understood from a more intuitive perspective:

In Figure.4, it seems that there are $P \cdot b$ frames processed simultaneously, but the obstacle followed is that all workers need to exchange the gradient matrix of very large dimensionality — N_M is estimated to be of the order of

10^8 (in Section 3.1) — with master twice (once for aggregation and once for redistribution), which is a big challenge to the bus bandwidth of server. For the limitation of the data transmission rate, transmission time take a large proportion of the whole processing time. Thus the training slows down because the clients would wait for each other.

4.6. ASGD vs. Synchronous data parallel

ASGD is fundamentally another form of data parallelism, where parameters read and written by worker nodes may be outdated by one iteration, in a non-determistic fashion. Rather, as discussed in Section 3.4, it is another form of delayed update. The above estimate of Eq.(8) also applies to ASGD with the only difference that ASGD allows for more flexible rounding where P is not an integer. This can reduce the effect of update delay by the order of b/P, but not more: If it did, one should add another node. The lesson here is that ASGD does not improve parallelizability in a fundamental way. If deterministic synchronized data parallel does not scale well, ASGD wonąŕt either.

5. Layer-Wise Model Partitioning

If we investigate the information flow of the BP procedure in more depth, we could notice that the interdependency among the variables happens more "inter-layer-ly" rather than "intra-layer-ly" (Figure. 5). Calculation within each layer is relatively independent and is attached only to the model parameters belonging to the same layer.

Hence, we are motivated to distribute the layers themselves across GPUs, in this way calculation could naturally be divided according the distribution of the model (Figure. 6). With the state vector σ^l (for forward step) and error vector B^l (backward) flow from GPU to GPU, the use of parameter server could be avoided to enable a better load balancing.

At present, most GPU devices support asynchronous transfer mode (ATM), which enables simultaneous data transmission and calculation. The extra cost

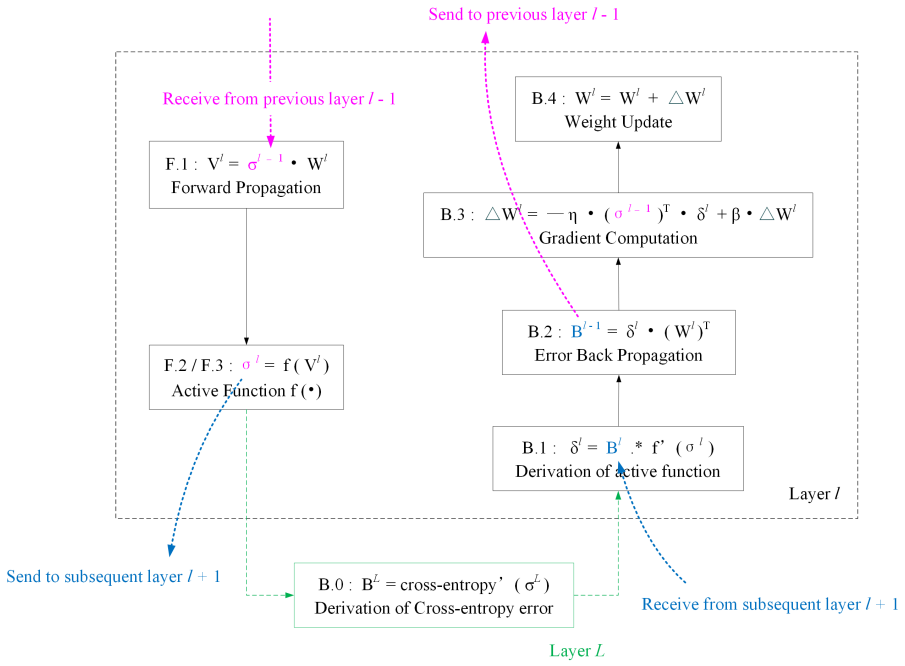


Figure 5: Data dependency of BP from a layer-wise perspective.

of communication overhead, introduced by computing task division and merging process, can be covered by continuous computation, enabling each node to work at their best with hardly any waste in waiting. This will lead to a better parallel efficiency as compared with ASGD, where each client has to spend a considerable part of time waiting for the parameter server for the latest model exchange (discussed in Section 4.5).

6. EXPERIMENTS

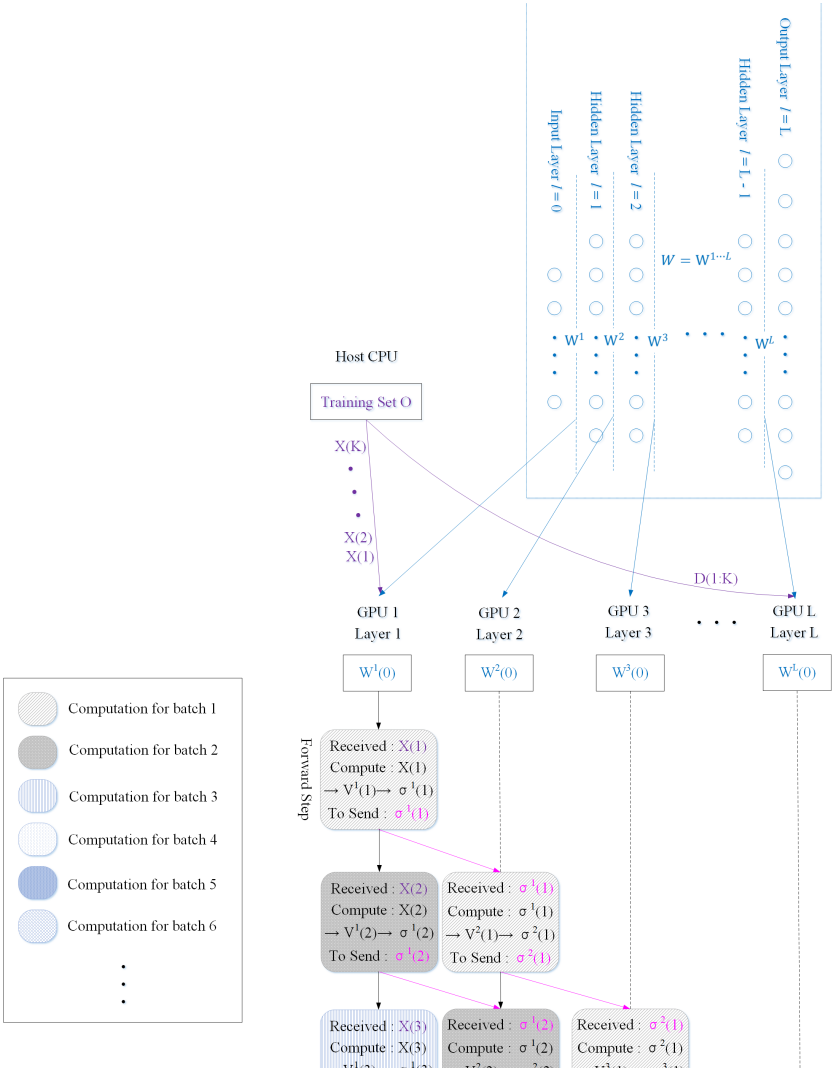
6.1. Hardware Configuration

Our main hardware is a Dell rack server equipped with 8 NVIDIA K20 GPU cards. Each K20 has 2096 cores with a peak speed of 3.52 $Tflops\ SP$ and 5 GB GDDR memory. The server is also configured with the Intel Xeon 2.7 GHz E5-2680 processor and 16 GB main memory.

6.2. Training data, model structure and feature extraction

We perform experiments on an English speech recognition benchmark, extracted from a standard US broadcast news shows[34]. After portions with spontaneous speech, noise, and background music are removed, it gives 142 hours of clean speech, publicly available at LDC[2]. 5% and 10% are randomly

[2] https://catalog.ldc.upenn.edu/LDC98S71



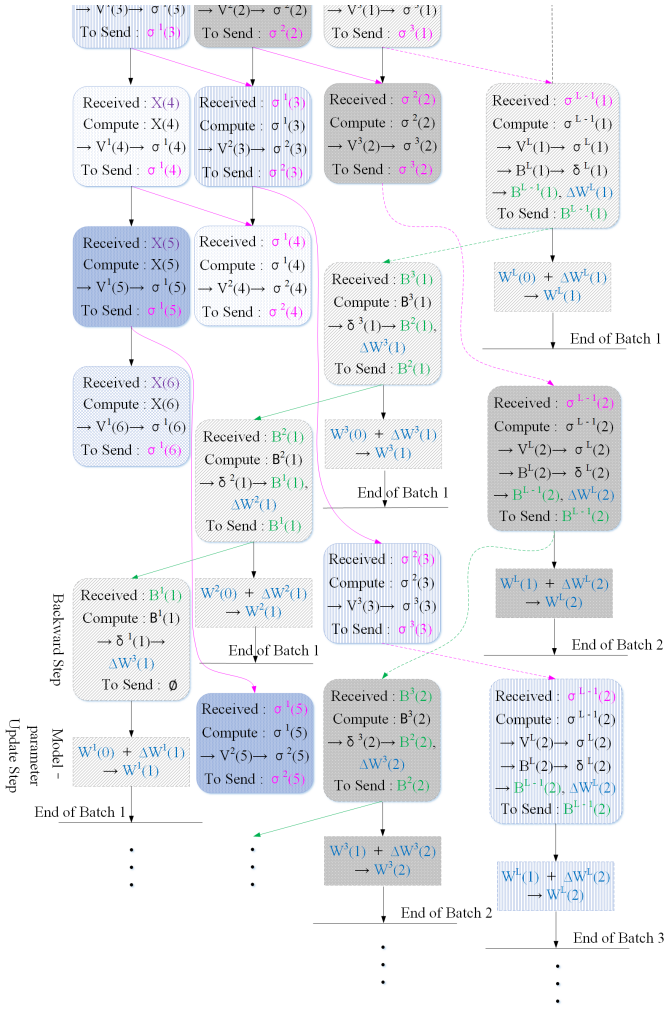


Figure 6: Training DNN under layer-wise model partitioning on multi-GPU.

held out for cross-validation and test purpose, respectively.

Acoustic feature vectors is extracted from every 10-ms signal frame. Each feature vector is 42-dimensional, formed by 13-dimensional PLP and pitch appended with their first and second order derivatives. Concatenations of 11 frames are used as input of our network which contains other 7 hidden layers with 2048 units and an output layer with 10217 senones.

Because DNN is trained at frame level, we need to generate training labels

Figure 7: The 8-GPU server used for neural network training in this work.

for each 10 ms frame from word-level transcriptions. A common approach is forced alignment under the guidance of a GMM-HMM baseline. we refer this pre-processing to [36] for detail.

6.3. Evaluation indicators

Unlike many other tasks, DNN in speech systems are not simply classifiers, but also serve as a sub-component who has to interact with its adjacent modules (the output of DNN serve as input to Viterbi searching module) in the pipeline. Since we view DNN mainly as a general classifier in this paper, frame classification accuracy (FAA) is our primary concern. But perfect classification at the level of short acoustic spans is not an ultimate goal of speech recognition systems. They wish to minimize the word error rate of the final system. Therefore, we report both DNN per-frame classifying performance (FAA) and final speech recognizer word error rate (WER) in our experiments. But it is necessary to note that WER is also affected by downstream HMM decoder, who encodes the word sequence probabilities from the language model.

6.4. Result and Discussion

Table 2 compares the performances and costs of different implementations.

Table 2: Test set performance after last training epoch and average training times in minutes per 10h of data.

	# GPU (P)	FAA (%)	WER (%)	Time	Speedup (S)	Efficiency (E)
GMM-HMM		—	25.79	—	—	—
Serial	1	66.25	18.50	195.1	—	—
Synchronous	4	63.97	18.57	130.0	1.5×	0.38
ASGD	4	63.92	18.58	63.0	3.1×	0.78
Layer-wise[1]	4	62.94	18.82	59.1	3.3×	0.83
Synchronous	8	63.75	18.72	92.9	2.1×	0.26
ASGD	8	63.80	18.75	57.4	3.4×	0.43
Layer-wise	8	62.39	18.87	28.7	6.8×	0.85

[1] When the number of nodes P is less than the number of network layers L, adjacent L/P layers are mapped to one node.

The first row shows results on the conventional GMM-HMM baseline system, the second row shows results trained by standard BP on single GPU. For ASGD and synchronous data parallel, one GPU is set as parameter server and the other $(P-1)$ cards serve as clients. For our proposed method, all P cards are engaged in actual minibatch computing.

Comparing with GMM, DNN achieves up to 30% reduction in terms of word error rate (WER). The per-frame accuracy of different implementations diverge only slightly, they all end up at around $18\% \sim 19\%$. This proves the equivalence of the 3 kinds of implementations as long as we apply the same mini batch policy (described in Section 3.4) during training iterations.

In term of the training time, asynchronous mode of SGD is always better than its synchronous counterpart, however they both face an early bottleneck when number of clients P reaches its upper bound of $\hat{P} = \frac{T_{calc}(b)}{2 \cdot T_{comm}} \approx 3$, if we continue to add GPU node to the system with more than $(\hat{P} + 1 = 4)$ nodes, it will not give further improvement to the speedup ratio, but will reduce the efficiency very obviously. This results are in agreement with the analytical

estimates we set forth in Section 4.4.

Our proposed method, on the other hand avoiding this bandwidth problem, is capable to achieve almost linear speedup with respect to the depth of the model, and gives a consistent efficiency above 80%.

7. CONCLUSION

In this paper, we described an effective approach to speed up training of fully connected DNN classifiers. The focus was on making best use of multiple GPUs inside a single compute server.

We compared the efficiency of distributing the training over dataset. For a fully connected DNN with $N_W = 45M$ parameters, data-parallel quickly runs into the bandwidth bottleneck: there is no benefit to use more than 4 GPUs as long as the standard minibatch size of 1024 is used; ASGD does not fundamentally change that.

We address this issue by a layer-wise model-parallel approach. Each GPU calculates gradients and updates the model parameters corresponding to the layers attributed to it. For multiple GPUs on a single server, this approach manages multiple GPUs to work effectively without waiting for each other.

Experimental results show that it achieves a 6.8 times speed-up on 8 GPUs than the single one, without noticeable loss in terms of WER, which is an 85% parallelization efficiency. This could help to cut the training time for very large collections (up to over 2000 hours to our knowledge[37]) from approximately a month [6] to 4 $\sim$ 5 days. Reducing the training time helps make many related research activity easier, such as enabling more adequate attempts in hyper-parameter tuning, testings of more various loss functions and optimization methods. We hope that this could serve as a useful reference in the now highly active research area of neural networks for speech and language understanding.

References

[1] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in neural information processing systems, 2012, pp. 1097–1105.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1–9. doi:10.1109/CVPR.2015.7298594.

[3] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.

[4] S. Ren, K. He, R. Girshick, J. Sun, Faster r-cnn: Towards real-time object detection with region proposal networks, IEEE Transactions on Pattern Analysis and Machine Intelligence PP (99) (2016) pre–print version. doi:10.1109/TPAMI.2016.2577031.

[5] K. He, X. Zhang, S. Ren, J. Sun, Spatial pyramid pooling in deep convolutional networks for visual recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence 37 (9) (2015) 1904–1916. doi:10.1109/TPAMI.2015.2389824.

[6] F. Seide, G. Li, D. Yu, Conversational speech transcription using context-dependent deep neural networks, in: 12th Annual Conference of the International Speech Communication Association (INTERSPEECH), 2011, pp. 437–440.

[7] G. E. Dahl, D. Yu, L. Deng, A. Acero, Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition, IEEE Transactions on Audio, Speech, and Language Processing 20 (1) (2012) 30–42. doi:10.1109/TASL.2011.2134090.

[8] L. Deng, G. Hinton, B. Kingsbury, New types of deep neural network learning for speech recognition and related applications: an overview, in: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 8599–8603. doi:10.1109/ICASSP.2013.6639344.

[9] D. Yu, L. Deng, Automatic Speech Recognition: A Deep Learning Approach, 1st Edition, Springer-Verlag London, 2014. doi:10.1007/978-1-4471-5779-3.

[10] Y. Huang, D. Yu, C. Liu, Y. Gong, A comparative analytic study on the gaussian mixture and context dependent deep neural network hidden markov models., in: 15th Annual Conference of the International Speech Communication Association (INTERSPEECH), 2014, pp. 1895–1899.

[11] J. Pan, C. Liu, Z. Wang, Y. Hu, H. Jiang, Investigation of deep neural networks (dnn) for large vocabulary continuous speech recognition: Why dnn surpasses gmms in acoustic modeling, in: The 8th International Symposium on Chinese Spoken Language Processing (ISCSLP), 2012, pp. 301–305. doi:10.1109/ISCSLP.2012.6423452.

[12] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, B. Kingsbury, Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, IEEE Signal Processing Magazine 29 (6) (2012) 82–97. doi:10.1109/MSP.2012.2205597.

[13] 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: Advances in Neural Information Processing Systems, 2012, pp. 1223–1231.

[14] T. Chilimbi, Y. Suzue, J. Apacible, K. Kalyanaraman, Project adam: Building an efficient and scalable deep learning training system, in: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 571–582.

[15] X. Lian, Y. Huang, Y. Li, J. Liu, Asynchronous parallel stochastic gradient for nonconvex optimization, in: C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett (Eds.), Advances in Neural Information Processing Systems 28 (NIPS), Curran Associates, Inc., 2015, pp. 2737–2745.

[16] Q. Meng, W. Chen, J. Yu, T. Wang, T. Liu, Asynchronous accelerated stochastic gradient descent, in: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI), 2016, pp. 1853–1859.

[17] S. Zhao, W. Li, Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee, in: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16), 2016, pp. 2379–2385.

[18] P. Moritz, R. Nishihara, I. Stoica, M. Jordan, Sparknet: Training deep networks in spark, in: Proceedings of the 4th International Conference on Learning Representations (ICLR), 2016. arXiv:1511.06051.

[19] H. Kim, J. Park, J. Jang, S. Yoon, Deepspark: Spark-based deep learning supporting asynchronous updates and caffe compatibility, in: Proceedings of the 22th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16), 2016. arXiv:1602.08191.

[20] R. Raina, A. Madhavan, A. Y. Ng, Large-scale deep unsupervised learning using graphics processors, in: Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09, ACM, New York, NY, USA, 2009, pp. 873–880. doi:10.1145/1553374.1553486.

[21] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, A. Y. Ng, Building high-level features using large scale unsupervised learning, in: J. Langford, J. Pineau (Eds.), Proceedings of the 29th International Conference on Machine Learning (ICML-12), ACM, New York, NY, USA, 2012, pp. 81–88.

[22] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, N. Andrew, Deep learning with cots hpc systems, in: Proceedings of the 30th International Conference on Machine Learning (ICML-13), 2013, pp. 1337–1345.

[23] G. Urban, K. J. Geras, S. E. Kahou, O. A. S. Wang, R. Caruana, A. Mohamed, M. Philipose, M. Richardson, Do deep convolutional nets really need to be deep (or even convolutional)?, in: Proceedings of the 4th International Conference on Learning Representations (ICLR), 2016.

[24] Z. Lin, R. Memisevic, K. Konda, How far can we go without convolution: Improving fully-connected networks, in: Proceedings of the 4th International Conference on Learning Representations (ICLR), 2016.

[25] D. Ciregan, U. Meier, J. Schmidhuber, Multi-column deep neural networks for image classification, in: Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, 2012, pp. 3642–3649. doi:10.1109/CVPR.2012.6248110.

[26] F. N. Iandola, K. Ashraf, M. W. Moskewicz, K. Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters [online] (2015). arXiv:1511.00175.

[27] H. Su, H. Chen. Experiments on parallel training of deep neural network using model averaging [online] (2015). arXiv:1507.01239.

[28] F. Rosenblatt, Principles of neurodynamics: perceptrons and the theory of brain mechanisms, Report (Cornell Aeronautical Laboratory), Spartan Books, 1962.

[29] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, Nature 323 (9) (1986) 533–536. doi:10.1038/323533a0.

[30] C. M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics), Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[31] J. Garofolo, et al. Timit acoustic-phonetic continuous speech corpus [online]. Linguistic Data Consortium, Philadelphia, 1993.

[32] The nvidia cuda basic linear algebra subroutines (cublas) library [online].

[33] CUDA Toolkit Documentation v7.5. Nvidia cuda compute unified device architecture programming guide [online].

[34] P. Woodland, T. Hain, G. Moore, T. Niesler, D. Povey, A. Tuerk, E. Whittaker, The 1998 htk broadcast news transcription system: Development and results, in: Proc. DARPA Broadcast News Workshop, 1999, pp. 265–270.

[35] 2001 Conversational Telephone Recognition Evaluation. National institute of standards and technology (nist) [online].

[36] B. Schuppler, S. Grill, A. Menrath, J. A. Morales-Cordovilla, Automatic phonetic transcription in two steps: Forced alignment and burst detection, in: L. Besacier, A.-H. Dediu, C. Martín-Vide (Eds.), Statistical Language and Speech Processing: Second International Conference, SLSP 2014, Grenoble, France, October 14-16, 2014, Proceedings, Springer International Publishing, Cham, 2014, pp. 132–143. doi:10.1007/978-3-319-11397-5_10.

[37] C. Cieri, D. Miller, K. Walker, The fisher corpus: a resource for the next generations of speech-to-text., in: LREC, Vol. 4, 2004, pp. 69–71.