Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 315–324

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster

Víctor Campos[†], Francesc Sastre[†], Maurici Yagües[†], Míriam Bellver[†], Xavier Giró-i-Nieto[§], and Jordi Torres[†§]

[†] Barcelona Supercomputing Center (BSC)
[§] Universitat Politècnica de Catalunya
{victor.campos,francesc.sastre,maurici.yagues,miriam.bellver,jordi.torres}@bsc.es
xavier.giro@upc.edu

Abstract

Deep learning algorithms base their success on building high learning capacity models with millions of parameters that are tuned in a data-driven fashion. These models are trained by processing millions of examples, so that the development of more accurate algorithms is usually limited by the throughput of the computing devices on which they are trained. In this work, we explore how the training of a state-of-the-art neural network for computer vision can be parallelized on a distributed GPU cluster. The effect of distributing the training process is addressed from two different points of view. First, the scalability of the task and its performance in the distributed setting are analyzed. Second, the impact of distributed training methods on the final accuracy of the models is studied.

Keywords: distributed computing; parallel systems; deep learning; Convolutional Neural Networks

1 Introduction

Methods based on deep neural networks have established the state-of-the-art in computer vision tasks [15, 27, 9], machine translation [29], speech generation [21] and even defeated the world champion in the game of Go [26]. Although the development of these algorithms spans over many decades [16], their potential has been unlocked by the increased computational power of specific accelerators, e.g. Graphical Processing Units (GPUs), and the creation of large-scale datasets [8, 13]. Even with the use of specific hardware devices, training these algorithms is so computationally intensive that can take days, or even weeks, to converge on a single machine. Scaling these problems to distributed settings that can shorten the training times has become a crucial challenge both for research and industry applications.

10.1016/j.procs.2017.05.074

Affective Computing [22] has recently garnered much research attention. Machines that are able to understand and convey subjectivity and affect would lead to a better human-computer interaction that is key in some fields such as robotics or medicine. Despite the success in some constrained environments such as emotional understanding of facial expressions [18], automated affect understanding in unconstrained domains remains an open challenge which is still far from other tasks where machines are approaching or have even surpassed human performance. In this work, we will focus on the detection of Adjective Noun Pairs (ANPs) using a large-scale image dataset collected from Flickr [12]. These mid-level representations, which are a rising approach for overcoming the *affective gap* between low level image features and high level affect semantics, can then be used to train accurate models for visual sentiment analysis or visual emotion prediction.

In this work, we explore how the training of Convolutional Neural Networks (CNNs) for ANP classification can be accelerated through distribution in a GPU cluster. Our contributions are three-fold: (1) we study the trade-off between final classification accuracy and speedup and analyze the results both from the learning and HPC standpoints, (2) distribute the training in a way that makes the most of the cluster resources, leveraging intra-node and inter-node parallelism, and (3) propose a modification of the distributed configuration, in order to reduce the resources being used while training.

2 Related Work

The massive number of convolutions and matrix multiplications in neural networks has led to GPU implementations with CUDA [20] and efficient, task specific primitives using cuDNN [7]. Early deep learning frameworks such as Caffe [10] provided fast and easy access to such primitives, but were initially designed for single machine operation, without support for distributed environments. Efforts towards distributing the former frameworks with traditional HPC tools such as Spark or MPI resulted in projects such as SparkNet [19] or Theano-MPI [17]. Native support for distributed settings is included in more recent frameworks such as TensorFlow [2] or MXNet [6]. However, scaling the training algorithms from a single machine environment to a distributed setting poses two main challenges. From the computing performance standpoint, optimizing the use of resources is the main goal, whereas from the learning side, the final accuracy should not suffer a drop when compared to its single machine counterpart.

We will consider the task of distributing Convolutional Neural Networks (CNNs), a specific type of feed-forward neural networks. CNNs are composed by a series of layers applying a specific operation to their input, e.g. convolution, dot product or pooling, and are trained to minimize a cost objective by means of gradient descent and backpropagation over batches of data. Two main approaches are proposed in the literature [14] to train CNNs on a multi-GPU environment, either in a single machine or in a distributed setting: model parallelism and data parallelism. *Model parallelism* splits layers in the CNN among different GPUs, i.e. each GPU operates over the same batch of input data, but applying different operations on them, and is mostly used for operations with a large number of parameters that may not fit in the GPU's memory. On the other hand, *data parallelism* consists in placing a replica of the model on each GPU, which then operates on a different batch of data. Model replicas share parameters, so that this method is equivalent to having a larger batch size. Modern CNN architectures aim to reduce the number of parameters while increasing the number of layers, finding a bottleneck when storing intermediate activations in memory. Unlike model parallelism, data parallelism only introduces one synchronization point regardless of the number of GPUs, thus reducing communication overhead and making it more suitable for current CNN architectures. Besides, balancing the load between GPUs is straightforward in this paradigm, while it would require from careful tuning for each specific CNN architecture and number of GPUs in a model parallelism approach. For these reasons, we will consider multi-GPU data parallelism for both single machine and distributed settings.

3 Dataset

Adjective Noun Pairs (ANPs) are powerful mid-level representations [3] that can be used for affect related tasks such as visual sentiment analysis or emotion recognition. A large-scale ANP ontology for 12 different languages, namely Multilingual Visual Sentiment Ontology (MVSO), was collected by Jou et al. [12] following a model derived from psychology studies, *Plutchick's Wheel of Emotions* [23]. The noun component in an ANP can be understood to ground the visual appearance of the entity, whereas the adjective polarizes the content towards a positive or negative sentiment, or emotion [12]. These properties try to bridge the *affective gap* between low level image features and high level affective semantics, which goes far beyond recognizing the main object in an image. Whereas a traditional object classification algorithm may recognize a *baby* in an image, a finer-grained classification such as *happy baby* or *crying baby* is usually needed to fully understand the affective content conveyed in the image. Capturing the sophisticated differences between ANPs poses a challenging task that benefits from leveraging large-scale annotated datasets by means of high learning capacity models [5].

In our experiments we consider a subset of the English partition of MVSO, the tag-restricted subset, which contains over 1.2M samples covering 1,200 different ANPs. Since images in MVSO were downloaded from Flickr and automatically annotated using their metadata, such annotations have to be considered as *weak labels*, i.e. some labels may not match the real content of the images. The tag-pool subset contains those samples for which the annotation was obtained from the tags in Flickr instead of other metadata, so that annotations are more likely to match the real ground truth.

4 CNN architecture

Since the first successful application of CNNs to large-scale visual recognition, the design of improved architectures for improved classification performance has focused on increasing the depth, i.e. the number of layers, while keeping or even reducing the number of trainable parameters. This trend can be seen when comparing the 8 layers in AlexNet [15], the first CNN-based method to win the Image Large Scale Visual Recognition Challenge (ILSVRC), with the dozens, or even hundreds, of layers in Residual Nets (ResNets) [9]. Despite the huge increase in the overall depth, a ResNet with 50 layers has roughly half the parameters in AlexNet. However, the impact of an increased depth is more notorious in the memory footprint of deeper architectures, which store more intermediate results coming from the output of each single layer, thus benefiting from multi-GPU setups that allow the use of larger batch sizes.

We adopt the ResNet50 CNN [9] on our experiments, an architecture with 50 layers that maps a $224 \times 224 \times 3$ input image to a 1,200-dimensional vector representing a probability distribution over the ANP classes in the dataset. Overall, the model contains over 25×10^6 single-precision floating-point parameters involved in over 4×10^9 floating-point operations that are tuned during training. It is important to notice that the more computationally demanding a CNN is, the larger the gains of a distributed training due to the amount of time spent doing parallel computations with respect to the added communication overhead.

Cross-entropy between the output of the CNN and the ground truth, i.e. the real class distribution, is used as loss function together with an L2 regularization term with a weight decay rate of 10^{-4}. The parameters in the model are tuned to minimize the former cost objective using batch gradient descent.

5 Distributed CNN training

The process of training CNNs with batch gradient descent can be decomposed in two main steps: forward and backwards passes through the net. The *forward pass* computes the outputs for a batch of data, and an error with respect to the desired result is then calculated. Such error, or *cost*, is then differentiated with respect to every parameter in the CNN during the so called *backwards pass*. Finally, the resulting gradients are used to update the weights in the net. These steps are iteratively repeated until convergence, i.e. until a local minima in the error function is reached.

In this implementation we use the *data parallelism* paradigm, in which we define two kind of nodes. First, the worker nodes each has a replica of the model, operating on separate batches of data. Second, the parameter server (PS) nodes store and update the model parameters [1]. In essence, the worker will receive the model parameters, compute it on a batch of data, and send back the gradients to the PS where the model will be updated in order to improve it [4]. However, different model update policies can be chosen at the PS to perform the training

Synchronous mode: In this case, the PS waits until all worker nodes have computed the gradients with respect to their data batches. Once the gradients are received by the PS, they are applied to the current weights and the updated model is sent back to all the worker nodes. This method is as fast as the slowest node, as no updates are performed until all worker nodes finish the computation, and may suffer from unbalanced network speeds when the cluster is shared with other users. However, faster convergence is achieved as more accurate gradient estimations are obtained. Authors in [4] present an alternative strategy for alleviating the slowest worker update problem by using backup workers.

Asynchronous mode: every time the PS receives the gradients from a worker, the model parameters are updated. Despite delivering an enhanced throughput when compared to its synchronous counterpart, every worker may be operating on a slightly different version of the model, thus providing poorer gradient estimations. As a result, more iterations are required until convergence due to the stale gradient updates. Increasing the number of workers may result in a throughput bottleneck by the communication with the PS, in which case more PS need to be added.

Mixed mode: mixed mode appears as a trade-off between adequate batch size and throughput by performing asynchronous updates on the model parameters, but using synchronously averaged gradients coming from subgroups of workers. Larger learning rates can be used thanks to the increased batch size, leading to a faster convergence, while reaching throughput rates close to those in the asynchronous mode. This strategy also reduces communication as compared to the pure asynchronous mode.

Others: improvements on the traditional gradient descent algorithm that can be applied to distributed settings have been proposed in the literature [31, 24, 25]. In this work, however, we focus on scaling problems from single node to distributed settings with minimal modifications to the training algorithm.

6 Experimental setup

We evaluate our experiments in a GPU cluster, where each node is equipped with 2 NVIDIA Kepler K80 dual GPU cards, 2 Intel Xeon E5-2630 8-core processors and 128GB of RAM. Inter-node communication is performed trough a 56Gb/s InfiniBand network. The CNN architectures and their training are implemented with TensorFlow[1], running on CUDA 7.5 and using cuDNN 5.1.3 primitives for improved performance. Since the training process needs to be submitted through Slurm Workload Manager, task distribution and communication between nodes is achieved with Greasy[2].

Unlike other works where each worker is defined as a single GPU [4, 11], we use all available GPUs in each node to define a single worker. Given the dual nature of the NVIDIA K80 cards, four model replicas are placed in each node. We follow the mixed approach to synchronously average the gradients for all model replicas in the same node before communicating with the PS, which then performs asynchronous model updates. This setup offers two main advantages: (1) communication overhead is reduced, as only a single collection of gradients needs to be exchanged through the network for each set of four model replicas, and (2) each worker has a larger effective batch size, providing better gradient estimations and allowing the use of larger learning rates for faster convergence.

The loss function is minimized using RMSProp [28] per-parameter adaptive learning rate as optimization method with a learning rate of 0.1, decay of 0.9 and $\epsilon = 1.0$. Each worker has an effective batch size of 128 samples, i.e. 32 images are processed at a time by each GPU. To prevent overfitting, data augmentation consisting in random crops and/or horizontal flips is asynchronously performed on CPU while previous batches are processed by the GPUs. The CNN weights are initialized using a model pre-trained on ILSVRC [8], practice that has been proven beneficial even when training on large-scale datasets [30].

Previous publications on distributed training with TensorFlow [1, 4] tend to use different server configurations for worker and PS tasks. Given that a PS only stores and updates the model and there is no need for GPU computations, CPU-only servers are used for this task. On the other hand, most of the worker job involves matrix computations, for which servers equipped with GPUs are used. All nodes in the cluster used in these experiments are GPU equipped nodes, which means that placing PS and workers in different nodes would result in under-utilization of GPU resources. We study the impact of sharing resources between PS and workers as compared to the former configuration and whether this setup is suitable for a homogeneous cluster where all nodes are equipped with GPU cards.

7 Results and discussion

7.1 Intra-node GPU parallelism

We first study the scalability of deploying multiple replicas with synchronous model updates on a single node of the GPU cluster. Due to the dual nature of the NVIDIA K80 cards, up to four model replicas can be deployed in every node. To ensure proper weight sharing between model replicas, the variables in the computation graph are stored in RAM, whereas each GPU performs all the operations of the CNN on a different batch of data. Gradients computed for each replica are averaged before performing the weights update, step that becomes the synchronization point in the graph.

[1] https://www.tensorflow.org/

[2] https://github.com/jonarbo/GREASY

Figure 2a shows the throughput as a function of the number of GPUs. We observe how we can speed up the training almost linearly with respect to the number of GPUs when performing synchronous updates, confirming the optimality of this policy for intra-node paralellism.

7.2 Throughput increase through distributed training

Parameter servers perform tasks that do not require from GPUs. However, given the homogeneus configuration of the cluster in which these experiments are performed, using nodes exclusively as PS would imply under-utilization of resources. In this section, we study the impact of sharing resources between PS and worker tasks.

Table 1 shows the trade-off between speedup and resource requirements when using 4 worker nodes with 4 GPUs each, following the mixed approach described in Section 5. Despite providing the largest speedup, the configuration where 3 dedicated PS are used requires from 7 nodes instead of 4. Should these additional GPU-equipped nodes be available, results in Figure 1 show how for a fixed number of nodes, solutions where all of them are used as workers provide the largest speedups. Given this figures, sharing resources between worker and PS tasks emerges as the most efficient solution for homogeneus GPU cluster such as the one considered in our experiments.

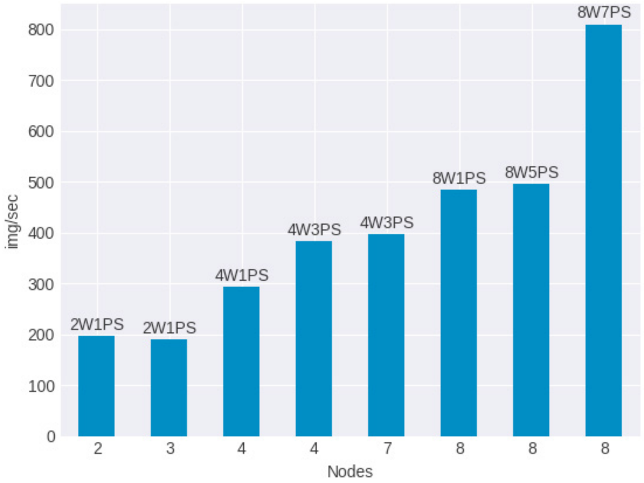


Figure 1: Throughput comparison between different distributed setups. Setting the proper number of parameter servers (PS) is key to maximizing throughput.

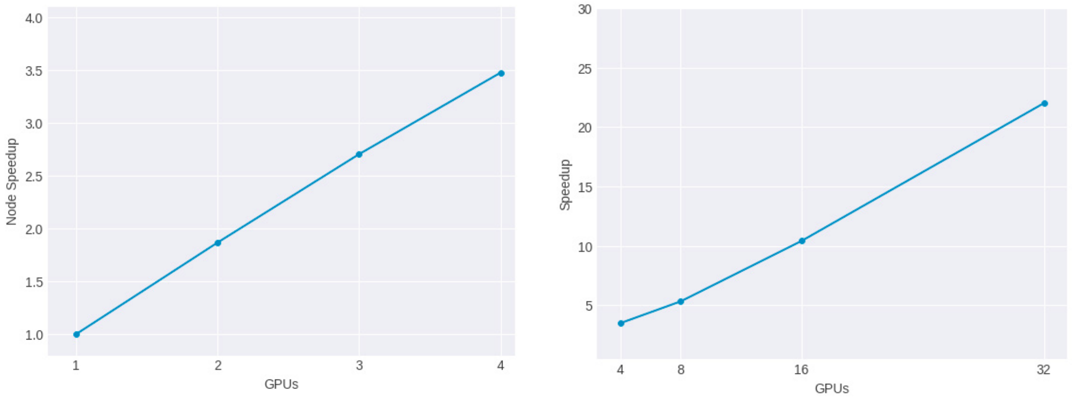
Results in Table 2 and Figure 2b show how the throughput speedup is close to linear for the configurations where resources are shared between PS and worker tasks. These figures demonstrate the scalability of the proposed approach while constraining the number of nodes required to achieve them.

7.3 Convergence speedup through distributed training

When studying the impact of distribution on the training process, there are two main factors to take into account. First, the time required for the model to reach a target loss value, which is the target function being optimized, determines which is the speedup on the training process.

Configuration	Throughput	Speedup	Efficiency
1 Node	124.18 img/sec	-	-
7 Nodes (4 Workers + 3 PS)	396.62 img/sec	3.19	0.46
4 Nodes (4 Workers + 3 PS)	374.73 img/sec	3.02	**0.76**
4 Nodes (4 Workers + 1 PS)	292.22 img/sec	2.35	0.58

Table 1: Comparison between different configurations when using 4 workers. Using dedicated nodes for the parameter servers slightly improves the throughput, but involves a much larger resource utilization. The efficiency is the relation between the speedup and the number of used nodes, it shows more clearly the grade of the resources exploitation.



(a)

(b)

Figure 2: Throughput speedup when using different number of resources. (a) Parallelism speedup inside a node using different number of GPUs. (b) Distribution speedup using nodes with 4 GPUs each, using the best configuration in Figure 1.

Second, the final accuracy determines how the asynchronous updates affect the optimization of the cost function.

Despite the throughput increase is close to linear with respect to the number of nodes, Figure 3 shows how the time required by each setup to reach a target loss value does not benefit linearly from the additional nodes. This result is expected, since for an asynchronous gradient descent method with N workers each model update is performed with respect to weights wich are $N-1$

Configuration	Throughput	Speedup	Efficiency
1 Node	124.18 img/sec	-	-
2 Nodes (2 Workers + 1 PS)	195.60 img/sec	1.58	0.79
4 Nodes (4 Workers + 3 PS)	383.09 img/sec	3.09	0.77
8 Nodes (8 Workers + 7 PS)	809.10 img/sec	6.52	0.82

Table 2: Throughput achieved by each distributed configuration. Speedup and efficiency figures are computed with respect to the 1 node scenario.

steps old on average.

The final accuracies on the test set reached by each configuration are detailed in Table 3. None of the distributed setups is able to reach the final accuracy of the single node model, confirming that the stale gradients have a negative impact on the final minima reached at which the model converges. Moreover, we found keeping a similar throughput between worker nodes to be a critical factor for a successful learning process, since workers that are constantly behind the rest of nodes do nothing but aggravate the stale gradients problem.

Workers (GPUs)	**Test Accuracy**	**Time (h)**	**Speedup**
1 Node (4)	0.228	106.43	1.00
2 Nodes (8)	0.217	62.78	1.69
4 Nodes (16)	0.202	37.99	2.80
8 Nodes (32)	0.217	22.50	4.73

Table 3: Results on the test set for the different distributed configurations. Despite benefitting from larger throughputs, setups with more nodes require more iterations to converge.

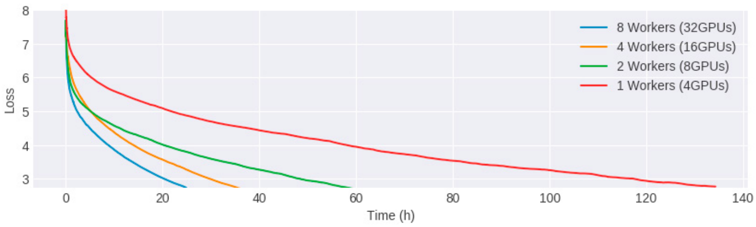


Figure 3: Train loss evolution for the different distributed configurations. The more nodes, the faster a target loss value is reached.

8 Conclusions and Future Work

Distributed training strategies for deep learning architectures will become more important as the size of datasets increases. They allow researchers to receive earlier feedback on their ideas and increase the pace at which algorithms are developed, thus understanding the best practices to distribute training of these models is a key research area. In our work, we studied how to adapt the training algorithm to the available hardware resources in order to accelerate the training of a CNN on a homogeneus GPU cluster. First, we showed how close to linear speedups can be achieved through intra-node parallelism. Based on these results, we developed a mixed approach where this efficiency is leveraged and the amount of inter-node communication is reduced as compared to a pure asynchronous policy. When properly tuning the number of parameter servers for each configuration, this method yields a important speedup in the number of samples per second processed by the system even for the setup with the minimum hardware overhead.

In spite of the good scalability demonstrated in terms of throughput, configurations with more nodes require from additional training steps to reach the same target loss value, although the increased throughput compensates this issue and still reduces the training time considerably.

This drawback becomes more important when increasing the number of nodes, so results suggest that different strategies should be employed for highly distributed settings with dozens of nodes.

Future work comprises two main research lines. First, the development of tools to gain more insight on the performance of each individual component that can help to detect bottlenecks and push even further the scalability of the system. On the other hand, we plan to implement and evaluate a pure synchronous gradient descent strategy. Despite solving the stale gradients problem, the overall throughput of this method is determined by the slowest worker, thus being less efficient than the mixed approach proposed in this work but using backup workers might be an option for achieve a great accuracy and maintain the performance. Besides, the increase in the effective batch size may have a negative impact on the generalization capabilities of the model, effect that will require further evaluation and experimentation.

Acknowledgements

This work is partially supported by the Spanish Ministry of Economy and Competitivity under contract TIN2012-34557, by the BSC-CNS Severo Ochoa program (SEV-2011-00067), by the SGR programmes (2014-SGR-1051 and 2014-SGR-1421) of the Catalan Government and by the framework of the project BigGraph TEC2013-43935-R, funded by the Spanish Ministerio de Economia y Competitividad and the European Regional Development Fund (ERDF). We also would like to thank the technical support team at the Barcelona Supercomputing center (BSC) especially to Carlos Tripiana.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. *ArXiv e-prints*, May 2016.

[2] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015.

[3] Damian Borth, Rongrong Ji, Tao Chen, Thomas Breuel, and Shih-Fu Chang. Large-scale visual sentiment ontology and detectors using adjective noun pairs. In *ACM MM*, 2013.

[4] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting Distributed Synchronous SGD. ICLR 2017 conference submission, 2016.

[5] Tao Chen, Damian Borth, Trevor Darrell, and Shih-Fu Chang. DeepSentiBank: Visual sentiment concept classification with deep convolutional neural networks. *arXiv:1410.8586*, 2014.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM MM*, 2014.

[11] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning? *arXiv preprint arXiv:1611.04581*, 2016.

[12] Brendan Jou, Tao Chen, Nikolaos Pappas, Miriam Redi, Mercan Topkara, and Shih-Fu Chang. Visual affect around the world: A large-scale multilingual visual sentiment ontology. In *ACM MM*.

[13] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *CVPR*, 2014.

[14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

[17] He Ma, Fei Mao, and Graham W Taylor. Theano-mpi: a theano-based distributed training framework. *arXiv preprint arXiv:1605.08325*, 2016.

[18] Daniel McDuff, Rana El Kaliouby, Jeffrey F Cohn, and Rosalind W Picard. Predicting ad liking and purchase intent: Large-scale analysis of facial responses to ads. *IEEE Transactions on Affective Computing*, 6(3), 2015.

[19] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[20] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[21] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[22] Rosalind W. Picard. *Affective Computing*, volume 252. MIT Press Cambridge, 1997.

[23] Robert Plutchik. *Emotion: A Psychoevolutionary Synthesis.* Harper & Row, 1980.

[24] S Sundhar Ram, Angelia Nedic, and Venugopal V Veeravalli. Asynchronous gossip algorithms for stochastic optimization. In *GameNets*, 2009.

[25] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*.

[26] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587), 2016.

[27] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[28] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

[29] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[30] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *NIPS*, 2014.

[31] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *NIPS*, 2015.