Accelerating Deep Learning with A Parallel Mechanism using CPU+MIC

Sijiang Fan, Jiawei Fei, and Li Shen

State Key Laboratory of High Performance Computing,
National University of Defense Technology, Changsha 410073, China
edwardhorp@gmail.com,lishen@nudt.edu.cn

Abstract. Deep neural networks(DNNs) is one of the most popular machine learning methods and is widely used in many modern applications. The training process of DNNs is a time-consuming process. Accelerating the training of DNNs has been the focus of many research works. In this study, we speed up the training of DNNs applied for automatic speech recognition and the target architecture is heterogeneous (CPU+MIC). We apply asynchronous methods for I/O and communication operations and propose an adaptive load balancing method. Besides, we also employ a momentum idea to speed up the convergence of the gradient descent algorithm. Results show that our optimized algorithm is able to acquire a 20-fold speedup on a CPU+MIC platform compared with the original sequential algorithm on a single-core CPU.

Keywords: CPU+MIC, Parallel Mechanism, DNNs, Deep Learning

1 Introduction

Deep learning has become the mainstream among various machine learning algorithms. In handwriting digital recognition[3], face detection[12], and automatic speech recognition (ASR)[2] fields, deep learning has outplayed other traditional machine learning algorithms like SVM[9].

The idea of deep learning originated in the 1980s. At that time, it was not very popular mainly due to a limited access to sufficient computational power. Nowadays, with the development of computational power, many achievements have been obtained in deep learning. Deeper networks and larger scale of datasets are adopted to achieve higher accuracy like the GoogLeNet, FaceNet[10]. These latest deep learning networks put forward higher requirements for computational power.

Recently, the many-core architectures have received significant attention in the acceleration research of deep learning. This is largely due to their inherent capability of parallel computing. Many-core architectures have two categories according to the types of cores (light cores and heavy cores). One is represented by NVIDIA and AMD GPUs which take advantage of massive light cores by fine-grained parallelism. The other is represented by Intel Many Integrated

Core (Intel MIC). Intel MIC has up to 60 heavy cores which support both fine-grained and coarse-grained parallelism. Many efforts for training deep networks on GPUs have been conducted and many deep learning frameworks such as Caffe[6], Torch7 and Tensorflow have supports for back-end training on GPUs. However, GPUs cannot work without CPU and they lack the ability to exploit coarse-grained parallelism due to hardware limits. Consequently, the independency and flexibility of GPUs are not as good as Intel MIC. MIC can not only work with CPU but also work independently and it supports coarse-grained parallelism. According to our survey, there are few deep learning studies with Intel MIC, let alone considering the combination of CPU and MIC. Most acceleration researches only focus on convolutional neural network (CNN) which is popular in image area.

In this paper, we aim to exploit fine-grained and coarse-grained parallelism for deep learning algorithms on the hybrid architecture with CPU and MIC. We focus on the ASR application using DNNs algorithm. Unlike the similar researches of deep learning on Intel MIC before, we focus on not only a single MIC but also the combination of CPU and MIC to achieve higher efficiency. To the best of our knowledge, our method which speeds up deep learning training by cooperative processing of both CPU and MIC is novel. Main contributions of this paper include,

- Parallelization approaches for supervised training of DNNs in ASR on the CPU-MIC hybrid architecture
- Asynchronous optimization for I/O and communication
- Adaptive load balancing between CPU and MIC

The experiments show that our optimized parallel DNNs training algorithm on the hybrid architecture with CPU and MIC gains 20-fold speedup compared with the serial code on a single-core Intel Xeon CPU.

The remaining of this paper is organized as follows, Sec.2 describes the basic concepts of DNNs and the Intel MIC architecture and introduces the related work so far. Sec.3 describes the general framework for DNNs training on hybrid architecture with CPU and MIC and introduces our optimization methods including the asynchronous optimization for I/O and communication, the implementation of adaptive load balancing between CPU and MIC and the momentum method to speed up algorithm convergence. The evaluation of our algorithm is presented in Sec.4. Sec.5 concludes our study and discusses our future work.

2 Related Work

In this section, we mainly introduce the work related to acceleration of deep learning algorithm on many-core architectures like GPUs and MIC.

On GPUs, Chetlur[1] created a library similar in intent to the Basic Linear Algebra Sub-routines (BLAS) which is called cuDNN and it improved the overall performance by 36% on a standard model when integrating it into Caffe. In the latest researches, NVidia launches an interactive system called Deep Learning

GPU Training System (DIGITS) and Jeffrey Dean[4] from Google presents a large-scale computer system for deep learning and accelerators of the system are GPUs. Since GPUs are good at matrix operation, accelerating CNN with GPUs is proved to have a very good performance. However, acceleration work on GPUs is not flexible due to their hardware limits and many deep learning algorithms like DNN, RNN and LSTM not only contain matrix operations. So the performance of accelerating these deep learning algorithms on GPUs is not always as good as CNN.

The Intel MIC architecture is a co-processor architecture which combines many Intel CPU cores onto a single chip and is capable of providing teraflops of double-precision and floating-point performance. Intel Xeon Phi consists of up to 61 Intel architecture cores, and each core supports 4 hardware threads. For many cores to access and communicate with each other, a bidirectional ring bus is designed. Moreover, the vector processing unit (VPU) is another feature that dictates the performance of applications on Xeon Phi. It supports 512 bit vector operations and implements a novel instruction set architecture.

Several studies related to deep learning have been carried out on Intel Xeon Phi. Jin[7] focused on the optimization of unsupervised learning of restricted Boltzmann machines and sparse auto-encoders that improved the performance 7 to 10 times on Intel Xeon Phi, compared with the CPU. Andre Viebke's[11] optimization of CNN decreased the training time from 31 h on CPU to 3 h on the Xeon Phi 7120p coprocessor. Junjie Liu[8] also achieved a speedup of 8.3 times on CNN with Intel Xeon Phi in his study. However, we find the studies so far mainly focus on single MIC in native mode which are not considering the combination of CPU and MIC. The work presented in this paper focuses on accelerating deep learning by combining CPU and MIC.

3 Parallel DNNs Algorithm On CPU and MIC Hybrid Architecture

3.1 General Parallel Framework

We designed a general parallel framework for DNNs training on the hybrid architecture with CPU and MIC, and focused on the automatic speech recognition (ASR) application using a general DNN model. In this section, we exploit the coarse-grained and fine-grained parallelism of the DNNs training algorithm on CPU+MIC architecture.

To exploit coarse-grained parallelism, we focus on sample-level parallelism. As we known, parallel DNNs training algorithm always employs a batch learning method to increase the parallelism. Samples in one batch are computed independently by different cores. In our research, we divide the batchs into two parts which are calculated individually by host (CPU) and MIC. Host and MIC will exchange the parameters of networks intermediately. The synchronization is performed as a result of averaging parameters on the host and MIC. The optimization methods for synchronization and load balancing are described in Sec.3.2 and Sec.3.3.

For fine-grained parallelism, we employ thread-level parallelism in many kernel functions by creating 4 threads on each MIC core and 2 threads on each CPU. We also take full use of VPUs on MIC to introduce vectorization for even better performance. The VPUs in Intel MIC provide data parallelism at a fine grain, working on 512 bits of 16 single-precision or 32 bit integers at a time. Besides, to take advantage of the cache on CPU and MIC, we employ cache-blocking method to improve data locality of program.The general framework of our parallel DNN algorithm is displayed in Fig.1.

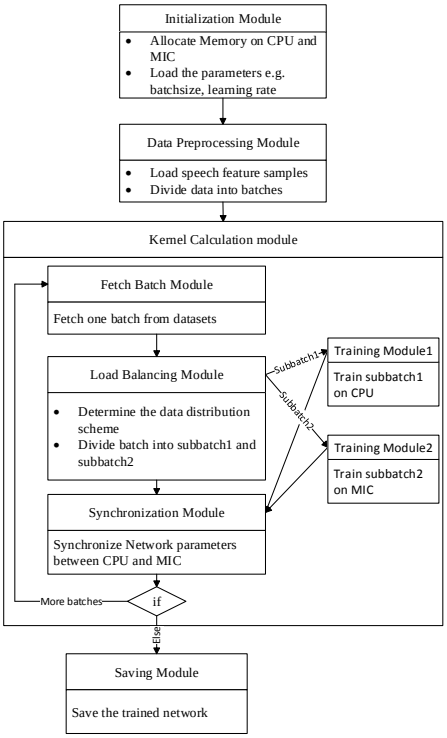

Fig. 1. The general framework of parallel DNNs algorithm on CPU+MIC architecture.

3.2 Asynchronous Optimization Methods

The Sec.3.1 describes the general framework. However, some factors constrain the performance of this framework. First, the training work of DNNs requires a large number of samples and the parameters of deep neural networks are always large-scale. So the I/O operation (loading samples and writing parameters) is one of the bottlenecks for big-data application like ASR. Another bottleneck which

affects the performance of parallel application on the hybrid architecture is the data transferring between host and cooperators. Because the communication between host and cooperators (MIC) is achieved through PCIE interface. In general, the bottlenecks mentioned above can be divided into two levels, Disk-to-Memory level and Memory-to-Memory (host to MIC) level. In this section, we propose asynchronous methods in both levels to optimize the I/O operations and data exchanging between host and MIC.

Loading and Calculation overlapped. In order to reduce the time consumption of data loading in Disk-to-Memory level, we employ an asynchronous method. In our method, two threads are created. One is responsible for loading data and the other for calculation. Fig.2 shows the procedure of asynchronous I/O. At first, the loading thread loads chunk i into buffer and turns to situation waiting for load signal from calculating thread. And then, the calculating thread moves the buffer data out and sets load signal for loading thread, at the same time, calculation for chunk i will be conducted. With the asynchronous I/O method, loading and calculating will be overlapped in a pipeline. The control of two threads is based on asynchronous signals supported by the pthread library.

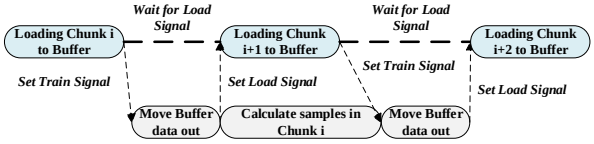

Fig. 2. Asynchronous method for I/O operation.

Calculation and Communication between the CPU and MIC in Parallel. An asynchronous method is also applied in Memory-to-Memory (host to MIC) level. We aim to reduce data transferring cost between CPU and MIC and allow them to calculate independently and simultaneously. The Implementation is slightly different from the method described above. Only one thread is created on the CPU and the signal mechanism is support by MIC's library. Fig.3 illustrates the implementation of the asynchronous method. After allocating load for CPU and MIC, the host will set the calculation signal. And the workload for MIC will be transferred from host memory to MIC's memory by MIC's support(#pragma offload_transfer). Calculation on CPU and MIC will be carried out simultaneously and the data transmission between CPU and MIC is hidden. After training of every batch, the MIC will set the synchronization signal and parameters will be synchronized between host and MIC.

Periodic Synchronization Algorithm. Although we apply asynchronous methods to reduce transferring cost between host and MIC, frequent data syn-

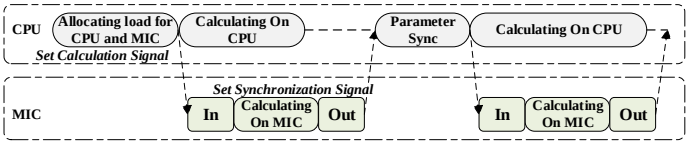

Fig. 3. Asynchronous method for calculating and communicating between CPU and MIC in parallel.

chronization still causes much extra time overhead. In our research, however, we find that appropriate reduction of synchronization frequency has little effect on the accuracy of DNNs model. In order to further reduce the communication overhead, we apply periodic synchronization algorithm. In this algorithm, host and MIC do not need to exchange temporary training results every batch. Instead, they will keep the temporary data individually in a period and conduct data synchronization every fixed number of batches. The periodic length determines algorithm's performance and accuracy.

3.3 Adaptive Load Balancing Algorithm

Load balancing is critical to improve performance for multi-thread applications. The key objective for load balancing is to minimize idle time on threads since an idle core during computation is a wasted resource and that will increase the overall execution time of a parallel application. However, achieving perfect load balance is non-trivial, especially for heterogeneous platforms like CPU+MIC architecture. Traditional static and dynamic load balancing are not suitable for the heterogeneous platforms because the computing capabilities of CPU and cooperators can hardly be measured and compared quantitatively in real time.

In this section, we propose an adaptive load balancing solution for parallel DNNs training focusing on the CPU+MIC hybrid architecture. In our solution, the proportion of workload assigned to CPU and MIC is adjusted according to the historical performance. We regard the number of samples trained per second as the performance measure and record it after training every batch. The load balance between CPU and MIC is achieved by dividing the batch dynamically during iterations.

Actually, the load balance problem can be abstracted as an optimal solution. Let ET as the execution time of entire DNNs training, TB_i as the execution time of the ith batch and T_{other} as the cost time of other operation like loading data or communication. ET can be computed by the formula,

$$ET = \sum_{i=1}^{N} TB_i + T_{other}.$$

The aim of the optimal problem is to minimize the ET value by adjusting the workload distribution between CPU and MIC. The allocation plan is expressed

with P which is a two-tuple $(x, 1-x)$ and x represents the percentage of workload assigned to CPU, $1-x$ represents the percentage of workload assigned to MIC. To simplify the problem, we assume that change of the allocation plan P has no effect on T_{other}. So the optimization goal is equivalent to minimizing value of each TB which is dependent on P and we can express TB with a function of pair $(x, 1-x)$,

$$TB_i = g(P_i) = g(x_i, 1 - x_i).$$

Then we adopt gradient descent method to find optimum allocation P. At first, we need to calculate gradient. We need to use the historical records to calculate gradient approximately in formula,

$$\nabla_i = \frac{\partial g}{\partial x} = \frac{\partial TB_i}{\partial x_i} \approx \frac{TB_i - TB_{i-1}}{x_i - x_{i-1}}.$$

Finally we can get the allocation P_{i+1} for the $i+1$th batch with formula,

$$P_{i+1} = (x_{i+1}, 1 - x_{i+1}) = (x_i - \gamma\nabla_i, 1 - x_i + \gamma\nabla_i).$$

In the formula, γ is the step length rate which controls the size of load migration when performing load balancing. Too small γ will lead to slow convergence while too large γ will lead to non-convergence, in other words, we will be unable to achieve the best allocation plan. Besides, to smooth the convergence and reduce accidental factors, we estimate the performance with multiple batches instead of single batch for gradient calculation.

3.4 Speed up Convergence with Momentum

To speed up convergence of DNNs training algorithm, we employ a technique using a momentum term[5]. It is a popular extension of the back propagation algorithm. With momentum m, the weight update at a given time t becomes,

$$\Delta w_{ij}(t) = \mu_i \delta_i y_i + m\Delta w_{ij}(t-1).$$

Where μ_i is learning rate, δ_i is relative error and $m(0 < m < 1)$ is a global parameter. Momentum method adds a fraction m of the previous weight update to the current one to smooth out the variations. When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum.

With momentum method, when using a lot of momentum, the learning rate needs to be reduced. In our method, we adapt the learning rate during training automatically to ensure the step size not to change too drastically. The learning rate is calculated by comparing Δw of Current moment and last moment, If the Δw of Current moment is larger, learning rate will decrease, otherwise, it will increase. The calculation process of μ_i is described as follows,

$$\mu_i = \mu_{i-1} \cdot \frac{|\Delta w_{t-1}|}{|\Delta w_t|}.$$

4 Evaluation

Experiments in this section are implemented on the Xeon Phi 31S1P with 57 cores and two Intel Xeon E5-2650 v2 CPU with 8 cores. The datasets we use are extracted from the Chinese and English speech database including 379904 speech samples, and the inputs of the DNNs are the feature vectors extracted from speeches. The DNNs network we used is an eight-layer network and the size of each layer is 429, $2048 * 6$, 8991. In order to verify the effectiveness of all methods proposed in this paper, we conduct five different experiments to evaluate our parallel DNNs algorithm focusing on different methods discussed earlier.

4.1 Performance of Asynchronous I/O

Firstly, we evaluate the performance of asynchronous I/O method(Disk-to-Memory) described in Sec.3.2. In the method, the data is loaded in chunks. To assess the impact of chunk size on the optimization method, we compare the accelerating effects under different chunk sizes. Fig.4 shows the change of execution time when the chunk size becomes larger. The time shown in the figure is obtained by running training algorithm multiple times and error is in 0.01 seconds.

As we can see from Fig.4, when the chunk size is too small (81920 samples), the performance of asynchronous I/O method is worse than the original serial I/O method. This is because the time overhead of thread creation and signal processing is larger than that of data transmission when the amount of data transmission is not large enough. And we also find when the amount of data transmission is too large, for example when the chunk size is 204800, the asynchronous method is equivalent to the serial method. It is obvious that the best chunk size value for our datasets is 102400 according to Fig.4.

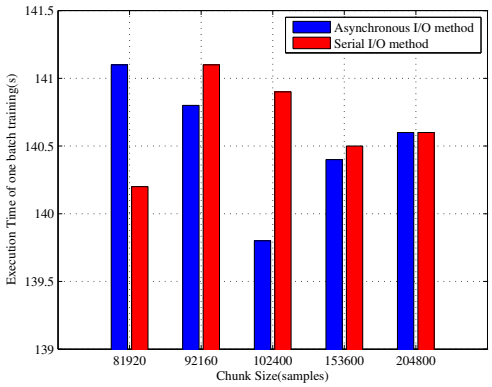

Fig. 4. The impact of chunk size on asynchronous I/O method(Disk-to-Memory).

4.2 Analysis of Periodic Synchronization Algorithm

In this sub-section, we evaluate the performance and accuracy of the periodic synchronization algorithm described in section 3.2.3 with different synchronization frequency. The synchronization interval varies from 2 batches to 100 batches in this experiment. In order to reduce the experimental time, we just choose part of the training set. As a result, the accuracy is relatively low. However, this will not disturb our assessment of acceleration effect. The results shown in this figure are obtained by multiple training and the error is about 0.01%.

As shown in Fig.5, when the synchronization interval is smaller than 30, the accuracy of the trained model remains generally stable and the execution time decreases obviously by almost 19%. When the synchronization interval becomes larger (> 30), the accuracy of model decreases and the execution time is no longer on the decline. This is a proof that appropriate reduction of synchronization frequency can improve performance with little effect on the accuracy of DNNs model.

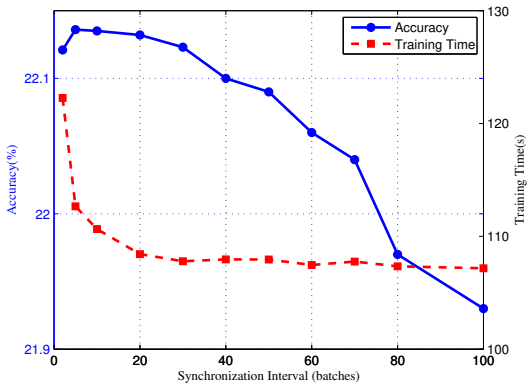

Fig. 5. The impact of synchronization interval on accuracy and performance.

4.3 Analysis of Adaptive Load Balancing

As described in Sec.3.3, for adaptive load balancing algorithm, it is critical to find the optimal load distribution quickly. In our experiment, the optimal load distribution will be obtained when the ratio of the workload allocated on CPU and MIC is $4 : 6$. The time cost on finding the optimal load distribution is controlled by the frequency of load distribution and the step length rate γ mentioned in Sec.3.3. In this sub-section, we analyze the impact of these two factors on the training performance, the performance is represented by the number of samples trained per second.

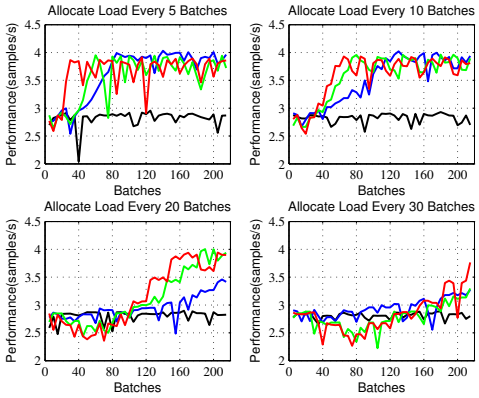

Fig. 6. The performance curve with different frequency of load distribution and γ (Black line: γ=0.1; Blue line: γ=0.2; Green line: γ=0.3; Red line: γ=0.4).

Fig.6 shows the performance curve during DNNs training with different frequencies of load distribution and γ. The frequency varies from once every 5 batches to once every 30 batches and the value of γ varies from 0.1 to 0.4. When the frequency of load distribution is high, the performance of training is unstable. And when it is low, the time cost on finding the optimal load distribution is much. Besides, no matter what frequency is, too small γ will cause the load balancing algorithm not to converge to the optimal load distribution while larger γ can bring faster convergence but may bring greater performance fluctuation. The experiment demonstrates that our adaptive load balancing algorithm can achieve load balance in a certain period of training time effectively and when the frequency is once every 10 batches and γ is 0.3, the performance is relatively good and stable.

We also tested the performance and accuracy under different allocation schemes. The results are displayed in Table.1 and reveal that training algorithm reaches peak performance when the ratio of the load allocated on CPU and MIC is $4 : 6$.

4.4 Analysis of Momentum

In this sub-section, we evaluate the effects of momentum on accelerating convergence. Fig.7 shows the accuracy convergence curve of DNNs training algorithm under different momentum values. It is obvious that with the increase of momentum value, the convergence of the algorithm is accelerated. When the value of momentum is 1.0, accuracy rate reaches the highest after only 3 epochs.

In this section, finally, we show the impact of each optimization method proposed in this paper on different hardware architecture including multi-core CPU, MIC and the hybrid architecture with CPU and MIC. The DNNs network we used is an eight-layer network and the size of each layer is 429, $2048 * 6$, 8991. Batch size we used is 1024 samples and Chunk size is 102400 samples.

Table 1. Performance and accuracy under different allocation scheme.

CPU:MIC	Accuracy	Time(s)	Performance(samples/s)
1:9	22.12	133.185	2852
2:8	22.09	120.790	3145
3:7	22.14	110.071	3451
4:6	22.12	**103.156**	**3682**
5:5	22.12	124.661	3047
6:4	22.16	142.955	2657
7:3	22.26	162.938	2331
8:2	22.29	177.219	2143
9:1	22.49	197.102	1927

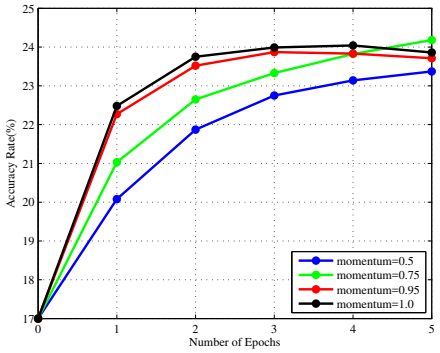

Fig. 7. The accuracy convergence curves with the increase of epoch number under different momentum values.

The baseline code is the serial code with no optimization methods and is run on the single-core CPU architecture. By employing different optimization techniques and porting to the many-core architecture, the training time decreases sharply. The result shown in Table.2 reveals that the fully optimized algorithm on hybrid architecture with CPU and MIC gains the best performance with approximately 20-fold speedup compared with a baseline code on a single-core CPU. The datasets used in our experiments are not very large, our algorithm will get better performance when the size of the dataset increases.

5 Conclusion

In this study, we propose a parallel algorithm for supervised DNNs training of ASR application focusing on the hybrid architecture with CPU and MIC. Compared with the sequential algorithm run on CPU, our optimized parallel algorithm gained approximately 20 speedup with a collaborative calculation of MIC and CPU. The experiments reveal that our optimization methods like asynchronous optimization and adaptive load balancing results in better performance.

Table 2. The overall results.

	Asyn_Methods	Load_Balance	Momentum_Algorithm	Time(s)
CPU	×	×	×	2050.3
Muti-core CPU	√	×	√	211.9
MIC	√	×	√	143.9
CPU+MIC	√	√	√	**103.2**

Acknowledgments. This work is supported in part by the National Natural Science Foundation of China under No.61472431. The authors would like to thank Chengkun Wu for his advising, and the anonymous reviewers for their time, work, and valuable feedback.

References

1. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. Computer Science (2014)
2. Chigier, B.: Automatic speech recognition (1997)
3. Cirean, D., Meier, U., Gambardella, L., Schmidhuber, J.: Deep, big, simple neural nets for handwritten digit recognition. Neural Computation 22(12), 3207–20 (2010)
4. Dean, J., Corrado, G.S., Monga, R., Chen, K., Devin, M., Le, Q.V., Mao, M.Z., Ranzato, A., Senior, A., Tucker, P.: Large scale distributed deep networks. Advances in Neural Information Processing Systems pp. 1232–1240 (2012)
5. Genevieve Orr, Nici Schraudolph, F.C.: Cs-449: Neural networks. https://www.willamette.edu/ gorr/classes/cs449/momrate.html (1999)
6. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. Eprint Arxiv pp. 675–678 (2014)
7. Jin, L., Wang, Z., Gu, R., Yuan, C., Huang, Y.: Training large scale deep neural networks on the intel xeon phi many-core coprocessor. In: IEEE International Parallel & Distributed Processing Symposium Workshops. pp. 1622–1630 (2014)
8. Liu, J., Wang, H., Wang, D., Gao, Y., Li, Z.: Parallelizing Convolutional Neural Networks on Intel ® Many Integrated Core Architecture. Springer International Publishing (2015)
9. Niranjan, M.: Support vector machines: a tutorial overview and critical appraisal. In: Applied Statistical Pattern Recognition (1999)
10. Pennycook, S.J., Hughes, C.J., Smelyanskiy, M., Jarvis, S.A.: Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors. In: Parallel and Distributed Processing Symposium, International. pp. 1085–1097 (2013)
11. Viebke, A., Pllana, S.: The potential of the intel (r) xeon phi for supervised deep learning. Computer Science pp. 758–765 (2015)
12. Zhang, C., Zhang, Z.: Improving multiview face detection with multi-task deep convolutional neural networks. In: IEEE Winter Conference on Applications of Computer Vision. pp. 1036–1041 (2014)