INTERSPEECH 2017
August 20–24, 2017, Stockholm, Sweden

Binary Deep Neural Networks for Speech Recognition

Xu Xiang, Yanmin Qian, Kai Yu

Key Lab. of Shanghai Education Commission for Intelligent Interaction and Cognitive Engineering
SpeechLab, Department of Computer Science and Engineering
Brain Science and Technology Research Center
Shanghai Jiao Tong University, Shanghai, China
{chinoiserie, yanminqian, kai.yu}@sjtu.edu.cn

Abstract

Deep neural networks (DNNs) are widely used in most current automatic speech recognition (ASR) systems. To guarantee good recognition performance, DNNs usually require significant computational resources, which limits their application to low-power devices. Thus, it is appealing to reduce the computational cost while keeping the accuracy. In this work, in light of the success in image recognition, binary DNNs are utilized in speech recognition, which can achieve competitive performance and substantial speed up. To our knowledge, this is the first time that binary DNNs have been used in speech recognition. For binary DNNs, network weights and activations are constrained to be binary values, which enables faster matrix multiplication based on bit operations. By exploiting the hardware population count instructions, the proposed binary matrix multiplication can achieve $5 \sim 7$ times speed up compared with highly optimized floating-point matrix multiplication. This results in much faster DNN inference since matrix multiplication is the most computationally expensive operation. Experiments on both TIMIT phone recognition and a 50-hour Switchboard speech recognition show that, binary DNNs can run about 4 times faster than standard DNNs during inference, with roughly 10.0% relative accuracy reduction.

Index Terms: speech recognition, binary deep neural networks, binary matrix multiplication, population count

1. Introduction

Deep neural networks (DNNs) have been shown to outperform Gaussian mixture models (GMMs) and have become the standard for acoustic modeling in speech recognition [1]. In recent large vocabulary continuous speech recognition systems, DNNs usually have six or more hidden layers with several thousands of neurons per layer. These models incur excessively high computational cost (mainly due to large matrix multiplication), which makes it infeasible to deploy such large models on low-power devices such as mobile phones or tablets. Hence, it is of great interest to speed up the calculation of DNNs while keeping the performance degradation in an acceptable range.

It has been reported that there is a significant redundancy in the parameterization of deep learning models [2], which leads to a waste of computation. Based on this phenomenon, different approaches have been proposed to reduce the redundancy. Xue *et al.* [3] applied singular value decomposition (SVD) on weight matrices to exploit the internal sparseness. Yu *et al.* [4] incorporated a soft regularization during training to minimize the number of nonzero elements of weight matrices. He *et al.* [5] and Qian *et al.* [6] proposed a method based on node pruning and arc restructuring to prune DNNs for fast inference. Han *et al.* [7] explored an iterative process to prune unimportant connections of DNNs, which reduced the storage and computation by an order of magnitude. Novikov *et al.* [8] represented weight matrices in a multi-linear format such that the number of parameters was largely reduced.

In addition to these approaches that are based on the transformation of matrices, the redundancy can also be reduced by quantization. Binary DNNs [9] is a recent technique that quantizes both network weights and activations to be binary. On a variety of image recognition benchmarks, it is able to gain significant acceleration during inference while still has competitive accuracies. Hence, it is attractive to validate the effectiveness of binary DNNs in speech related tasks.

In this paper, the use of binary DNNs in speech recognition is intensively studied. The contributions of this work are as follows: 1) it is the first time that binary DNNs are used in speech recognition; 2) the proposed binary matrix multiplication implementations that can run 5-7 times faster than aggressively optimized floating-point baselines on Intel, ARM CPUs and NVIDIA GPUs; 3) detailed comparisons between binary DNNs and standard DNNs in speed and accuracy on two speech recognition tasks.

The rest of the paper is organized as follows. Section 2 describes the architecture of binary DNNs. Section 3 analyzes the theoretical peak performance of floating-point and binary matrix multiplication on specific hardware, then compares the real performance between the proposed binary implementations and the state-of-the-art floating-point implementations. Experimental setup and results on TIMIT phone recognition and Switchboard conversational speech recognition are given in Section 4. Finally, Section 5 concludes the whole paper.

2. Binary DNNs

This section gives the algorithmic description of binary DNNs and insights on the optimization for fast inference.

2.1. Binarization of weights and activations

In a feedforward neural network with L layers, let the activation in layer l be a_l, the weight matrix between layer l and $l+1$ be $W_{l+1,l}$, and the bias in layer $l+1$ be b_{l+1}, where $1 \leq l \leq L-1$. The binarized weight matrices and activations are denoted by $\hat{W}_{l+1,l}$ and $\hat{a}_l$. Algorithm 1 describes the forward pass of binary DNNs. Note that weight matrix $W_{2,1}$ is not binarized, since a_1 (i. e. the network input) can not be properly binarized or quantized.

In Algorithm 1, function $\mathrm{Binarize}(\cdot)$ is used to transform each element of $W_{l+1,l}$ or a_l to $+1$ or -1. The deterministic binarization function is defined by

$$\text{Binarize}(x) = \begin{cases} +1, & \text{if } x > 0 \\ -1, & \text{otherwise} \end{cases} \quad (1)$$

where x is a floating-point value. The stochastic version that used in this work is different from the one in [9], which is defined by

$$\text{Binarize}(x) = \begin{cases} +1, & \text{if } x - p > 0 \\ -1, & \text{otherwise} \end{cases} \quad (2)$$

where p is a random value that drawn from a normal distribution with zero mean and unit variance. Stochastic binarization is more computationally expensive than the deterministic version, but it reduces overfitting, hence it is used to binarize the activations during training. Note that although function $\text{HardTanh}(x) = \max(-1, \min(x, 1))$ takes no effect in the forward pass since $\text{Binarize}(\text{HardTanh}(\cdot)) = \text{Binarize}(\cdot)$, it plays an important role in the backward pass.

Algorithm 1 Forward pass of binary DNNs

```
Input: input (which is also the activation of input layer) $a_1$,
    weight matrices $W_{2,1}, \cdots, W_{L,L-1}$, and biases $b_2, \cdots, b_L$
Output: activations $a_2, \cdots, a_L$

1. input layer: $W_{2,1}$ is not binarized
$a_2 = W_{2,1} \times a_1 + b_2$
$a_2 = \text{BatchNorm}(a_2)$
$a_2 = \text{HardTanh}(a_2)$
$\hat{a}_2 = \text{Binarize}(a_2)$

2. hidden layers: $W_{l,l-1}$ and $a_l$ $(3 \le l \le L-1)$ are binarized
for $l = 3$ to $L - 1$ do
    $\hat{W}_{l,l-1} = \text{Binarize}(W_{l,l-1})$
    $a_l = \hat{W}_{l,l-1} \times \hat{a}_{l-1} + b_l$
    $a_l = \text{BatchNorm}(a_l)$
    $a_l = \text{HardTanh}(a_l)$
    $\hat{a}_l = \text{Binarize}(a_l)$
end for

3. output layer: $a_L$ is not binarized
$\hat{W}_{L,L-1} = \text{Binarize}(W_{L,L-1})$
$a_L = \hat{W}_{L,L-1} \times \hat{a}_{L-1} + b_L$
$a_L = \text{BatchNorm}(a_L)$
$a_L = \text{Softmax}(a_L)$
```

2.2. Straight through estimator

While the forward pass is straightforward, there is an issue in the backward pass: mathematically the gradient of function $\text{Binarize}(\cdot)$ is always zero with respect to its input, which makes the gradient based training impossible. However, this can be resolved by using "straight through estimator" (STE) [10]. Here an variant of STE appeared in [9] is used, which cancels the gradient when the magnitude of the input is too large:

$$\begin{aligned} &\text{forward: } q = \text{Binarize}(p) \\ &\text{backward: } \frac{\partial\,\text{loss}}{\partial p} = \frac{\partial\,\text{loss}}{\partial q} \times \mathbb{1}_{|p| \le 1} \end{aligned} \quad (3)$$

The indicator function $\mathbb{1}_{|p| \le 1}$ is exactly the gradient of function $\text{HardTanh}(\cdot)$. Thus technically, in the backward pass, function $\text{Binarize}(\cdot)$ can be treated as an identity function.

2.3. Optimization for fast inference

2.3.1. Module reforming

Batch normalization [11] used in Algorithm 1 can be described as

$$\text{BatchNorm}(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \quad (4)$$

where x is the layer input, γ and β are the learnable parameters, and ϵ is a small value to avoid underflow. During inference, the mean μ and variance σ^2 are replaced with fixed values that are estimated over the training data, which can lead to an efficient and compact representation:

$$\text{BatchNorm}(x) = \xi x + \delta \quad (5)$$

where $\xi = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}$ and $\delta = \beta - \frac{\mu\gamma}{\sqrt{\sigma^2+\epsilon}}$, both can be precomputed before the deployment.

2.3.2. Module compacting

During training, floating-point weights are essential to do the update, but during inference, only their binarized version are utilized. For this reason, by replacing $W_{l,l-1}$ with $\hat{W}_{l,l-1}$ in Algorithm 1, module $\hat{W}_{l,l-1} = \text{Binarize}(W_{l,l-1})$ can be removed. Moreover, considering $\text{Binarize}(\text{HardTanh}(\cdot)) = \text{Binarize}(\cdot)$, module $a_l = \text{HardTanh}(a_l)$ can also be removed. Furthermore, module $a_l = \text{BatchNorm}(a_l)$ and $\hat{a}_l = \text{Binarize}(a_l)$ can be seamlessly integrated into one module so that the unnecessary computation can be avoided.

3. Binary matrix multiplication

This section describes population count based binary matrix multiplication, analyzes its performance gain and compares it with floating-point matrix multiplication in both theory and practice.

3.1. Population count based binary matrix multiplication

In practical applications, the multiplication of two matrices $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$, requires $2 \times m \times n \times k$ floating-point arithmetic operations (i. e. multiplications and additions)[1]. Due to hardware limitation of these operations, the speed of floating-point matrix multiplication is hard to improve.

However, in the binary case, it is possible to replace most multiplications and additions with bit operations, as each element of the resulting matrix can be represented by an inner product: $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$. To explain this in detail, two bit operations xor and popcnt are defined as: 1) $\text{xor}(x, y)$ is the element-wise exclusive or of integer x and y; 2) $\text{popcnt}(x)$ is the number of bits set to 1 in integer x. Assuming there are two vectors a and b with length n, that their elements are constrained to be either $+1$ or -1 (binary values). Such representation of a or b is not suitable for fast computation, so an extra processing step is needed. First, all -1s are replaced with 0s, then the n 0,1 bits are packed into $\lceil n/k \rceil$ ($k = 32$ or 64) k-bit integers. This process converts a and b to $\bar{a}$ and $\bar{b}$. The inner product of a and b can be calculated as

$$\langle a, b\rangle = n - 2 \times \text{popcnt}(\text{xor}(\bar{a}, \bar{b})) \quad (6)$$

[1]Strassen's algorithm runs in $O(n^{\ln 7}) = O(n^{2.81})$ time. However, it has a large hidden constant factor and is extremely cache unfriendly, which makes it difficult to get good performance on contemporary processors.

For a better understanding, an example is given here. Let a and b be two vectors of length 8: $a = (1, -1, 1, 1, 1, 1, 1, 1)$ and $b = (-1, 1, 1, -1, -1, 1, -1, 1)$. It is easy to find that their inner product is -2, which is equal to $8 - 2 \times \text{popcnt}(\text{xor}(\bar{a}, \bar{b})) = 8 - 2 \times \text{popcnt}(\text{xor}(10111111_2, 01100101_2)) = 8 - 2 \times \text{popcnt}(11011010_2) = 8 - 2 \times 5$.

3.2. Binary matrix multiplication on CPU

Recent CPUs have built-in support for population count instructions for both 32-bit and 64-bit operands. On Intel Haswell microarchitecture, two 8-wide FMA (fused multiply-add) instructions can be performed every cycle, yielding 32 32-bit floating-point operations per cycle [12]. By contrast, the 64-bit population count instruction can be issued every cycle, yielding 128 binary operations per cycle (other instructions like xor can be issued simultaneously due to superscalar execution). In other words, the population count instruction based binary matrix multiplication has $4.0 = 128/32$ times throughput compared with optimized floating-point matrix multiplication. On ARM Cortex A72 microarchitecture, one 4-wide FMA instruction, or up to 8 floating-point operations can be performed per cycle. Although the precise timing of population count instruction on ARM can not be estimated, empirical evaluation shows the speed up it delivers is considerable.

To achieve compute-bound performance, the implementation of binary matrix multiplication uses packing to ensure consecutive memory locations access and cache- and register-aware blockings to maximize data reuse. For more details, please refer to [13, 14].

An Intel i3-4150 CPU (Intel Haswell microarchitecture) running at 3.50 GHz and a HiSilicon Kirin 950 CPU (ARM Cortex A72 microarchitecture) running at 2.30 GHz were used to compare the speed between binary matrix multiplication and floating-point matrix multiplication. The baseline floating-point implementation utilized Intel Math Kernel Library 11.3 Update 3 on Intel platform and OpenBLAS 0.2.19 on ARM platform to achieve maximum speed. To determine the maximum real performance, a matrix multiplication of size $(m, n, k) = (2048, 2048, 2048)$ is commonly used. It is worth noting that, in practice, the batch size m is much smaller than 2048 during inference. Hence, the performance of the matrix multiplication of size $(16, 2048, 2048)$ corresponding to $m = 16$ was also evaluated.

Table 1 reports the single thread GOPS[2] (i. e. billions of floating-point/binary arithmetic operations, or ops, per second) on an Intel i3-4150 CPU. It is shown that, when batch size is 16, binary matrix multiplication can achieve 7.2× speed up. Table 2 reports single thread GOPS on a HiSilicon Kirin 950 CPU. Similar to the results of Table 1, binary matrix multiplication is 6.7× faster when batch size is 16.

Table 1: *Speed comparison on an Intel i3-4150 CPU (single thread).* **FMM***: floating-point matrix multiplication.* **BMM***: binary matrix multiplication.* **TPP***: theoretical peak performance*

Size	FMM	BMM	Speedup
$16, 2048, 2048$	34.5	249.0	$\mathbf{7.2}\times$
$2048, 2048, 2048$	91.2	263.5	$2.9\times$
TPP	112.0	448.0	$4.0\times$

[2]For each matrix size, the calculation was repeated 100 times to obtain the average GOPS.

Table 2: *Speed comparison on a HiSilicon Kirin 950 CPU (single thread).* **FMM***: floating-point matrix multiplication.* **BMM***: binary matrix multiplication.* **TPP***: theoretical peak performance*

Size	FMM	BMM	Speedup
$16, 2048, 2048$	3.8	25.4	$\mathbf{6.7}\times$
$2048, 2048, 2048$	12.0	29.1	$2.4\times$
TPP	18.4	≈ 39.5	$2.1\times$

3.3. Binary matrix multiplication on GPU

Population count instructions are natively supported by recent NVIDIA GPUs via intrinsics, including `popc__()` (an intrinsic function that maps to a single instruction) for 32-bit operands and `popcll__()` (an intrinsic function that maps to a few instructions) for 64-bit operands.

On NVIDIA Pascal microarchitecture Tesla P100 GPU, up to 64 32-bit FMA instructions (i. e. 128 floating-point operations) can be issued every cycle per streaming multiprocessor. In contrast, 16 32-bit population count instructions, 64 32-bit integer add instructions, or 64 32-bit bitwise xor instructions can be issued every cycle per streaming multiprocessor, yielding $683 = 2 \times 64 \times 32/(1 + 4 + 1)$ binary operations per cycle. Hence, in theory, binary matrix multiplication can achieve $5.3 = 683/128$ times speed up.

The implementation of binary matrix multiplication uses a modified version of an example program in CUDA toolkit document[3] that replaces multiply-add operations with bit operations. The floating-point baseline utilized NVIDIA cuBLAS library to achieve best performance.

Table 3 reports the GOPS of both implementations. When batch size is 16, the proposed binary matrix multiplication is able to achieve $5.4\times$ speed up.

Table 3: *Speed comparison on NVIDIA Tesla P100 GPU.* **FMM***: floating-point matrix multiplication.* **BMM***: binary matrix multiplication.* **TPP***: theoretical peak performance*

Size	FMM	BMM	Speedup
$16, 2048, 2048$	1.3×10^3	7.0×10^3	$\mathbf{5.4}\times$
$2048, 2048, 2048$	7.6×10^3	30.6×10^3	$4.0\times$
TPP	9.3×10^3	49.6×10^3	$5.3\times$

4. Experiments and results

This section describes the experiments on TIMIT phone recognition task and a 50-hour Switchboard speech recognition task. The speed and recognition accuracy between standard DNNs and binary DNNs on two tasks are compared. All DNN models were trained using Torch7 [15]. Classical GMM-HMM models that generate the alignments for DNN training were trained by Kaldi [16]. Kaldi was also used to decode the speech.

4.1. Experiments on TIMIT phone recognition task

The TIMIT corpus of read speech contains 4288 sentences spoken by 630 speakers selected from 8 major dialect regions of American English. The training set contains 3696 sentences from 462 speakers. The development set contains 400 sentences

[3]http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory

from 50 speakers and the test set contains 192 sentences from 24 speakers. 95% of the training set sentences were used as training data and the remaining 5% were used as validation data. Recognition accuracies were reported on the development set and test set.

First, 13-dimensional mel-frequency cepstral coefficients (MFCC) features were extracted with per speaker cepstral mean normalization (CMN), then spliced in time with a context of ± 3 frames, and projected to 40 dimensions with linear discriminant analysis (LDA). The resulting features were de-correlated using maximum likelihood linear transform (MLLT) and then applied speaker normalization by feature space maximum likelihood linear regression (fMLLR).

The input vector was constructed by concatenating 11 consecutive frames (a central frame with ± 5 contexts) of 40-dimensional fMLLR features, and the one-hot encoding target vector corresponding to the central frame was generated by state level forced alignment of a GMM/HMM model with 1947 tied triphone states. To classify the central frame, a model structure consisted of an input layer with 440 units, 6 hidden layers with 1024 units per layer, and an output layer with 1947 units was used. ReLU was used as the non-linear function in hidden layers of standard model. Cross entropy (CE) was used as training criterion. Stochastic gradient descent (SGD) was used to train the standard model, with an initial learning rate of 0.1, while AdaMax [17] was used to train the binary model, with an initial learning rate of 0.001. During training, the batch size was 256.

Both two models were evaluated on an Intel i3-4150 CPU[4]. During inference, the batch size was set to 16. Phone error rate (PER) and frames per second (FPS) are shown in Table 4.

Table 4: *PER (%) and FPS on TIMIT phone recognition task.* **SDNN***: standard DNN.* **BDNN***: binary DNN*

Model	dev	test	FPS
SDNN	17.6	18.7	2440
BDNN	19.0	20.1	9900

It is observed that although the first layer is not binarized, in the forward pass the binary model is still $4.0\times$ faster than the highly optimized standard model. The accuracy gap between the two models is not very big, since the relative PER increments of binary model are only 7% on both sets.

4.2. Experiments on 50-hour Switchboard speech recognition task

For fast development, a subset of Switchboard corpus was used in this experiment[5]. The subset contains 50-hour audio data spoken by 810 speakers that randomly chosen from the whole 309-hour Switchboard dataset. The Switchboard/CallHome (refered to as SWB and CH) portion of the NIST Hub5 2000 evaluation set and the Fisher/Switchboard (refered to as FSH and SWB) portion of the Rich Transcription 2003 evaluation set were used as the test sets.

36-dimensional log mel-frequency filter bank (FBANK) along with their first and second order derivatives were extracted, and per speaker CMN was applied to the features.

[4]Matrix multiplications account for more than 90% of overall computation, thus other factors like using different non-linear functions have small impact on FPS.

[5]The experiments on the full Switchboard data set are still running when submitting this manuscript, so only the results on the subset are reported here.

The input was formed of 11 consecutive frames of those acoustic features, and the corresponding target was built by state level forced alignment of a GMM-HMM model with 2723 tied triphone states. The model structure had 6 hidden layers with 2048 units per layer. Sigmoid was used as non-linear function in hidden layers of standard model. The training criterion, learning rate and batch size were the same as those of TIMIT task. A trigram language model trained on the Switchboard transcripts was used for decoding. Word error rate (WER) was used as performance measure. During inference, the batch size was set to 16. Table 5 shows the results of standard model and binary model that were evaluated on an Intel i3-4150 CPU.

Table 5: *WER (%) and FPS on Switchboard word recognition task.* **SDNN***: standard DNN.* **BDNN***: binary DNN*

Model	Hub5'00		RT03S		FPS
	SWB	CH	FSH	SWB	
SDNN	20.9	35.4	26.7	37.3	700
BDNN	23.6	38.1	29.6	39.8	2590

Similar to the results of TIMIT task, the proposed binary model achieves substantial $3.7\times$ speed up compared with the standard model. On four test sets (SWB/CH of Hub5'00 and FSB/SWB of RT03S), the performance of the binary model is slightly worse than the standard model: the relative increments of WER are 12.9%, 7.6%, 10.9%, and 6.7% respectively.

5. Conclusions

This paper presents detailed work on using binary DNNs for fast inference in speech recognition. First, based on bit operations such as population count, efficient binary matrix multiplication is proposed on specific hardware. In comparison with highly optimized floating-point baseline, the implementation can achieve substantial $5 \sim 7$ times speed up when the corresponding "batch size" is 16. As matrix multiplications account for more than 90% of overall computation, DNNs with binary weights and activations can benefit a lot from this acceleration. Then, the proposed binary model is evaluated on TIMIT phone recognition task and a 50-hour Switchboard speech recognition task to verify the effectiveness of binary DNNs. The results show that, the binary model can run about 4 times faster than the standard model during inference, with only around relative 10.0% accuracy reduction.

6. Acknowledgements

This work was supported by the Shanghai Sailing Program No. 16YF1405300, the China NSFC projects (No. 61573241 and No. 61603252) and the Interdisciplinary Program (14JCZ03) of Shanghai Jiao Tong University in China. Experiments have been carried out on the PI supercomputer at Shanghai Jiao Tong University.

7. References

[1] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[2] M. Denil, B. Shakibi, L. Dinh, N. de Freitas *et al.*, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, 2013, pp. 2148–2156.

[3] J. Xue, J. Li, and Y. Gong, “Restructuring of deep neural network acoustic models with singular value decomposition.” in *Interspeech*, 2013, pp. 2365–2369.

[4] D. Yu, F. Seide, G. Li, and L. Deng, “Exploiting sparseness in deep neural networks for large vocabulary speech recognition,” in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE, 2012, pp. 4409–4412.

[5] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu, “Reshaping deep neural network for fast decoding by node-pruning,” in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 245–249.

[6] Y. Qian, T. He, W. Deng, and K. Yu, “Automatic model redundancy reduction for fast back-propagation for deep neural networks in speech recognition,” in *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015, pp. 1–6.

[7] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in *Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[8] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, “Tensorizing neural networks,” in *Advances in Neural Information Processing Systems*, 2015, pp. 442–450.

[9] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1,” *arXiv preprint arXiv:1602.02830*, 2016.

[10] Y. Bengio, N. Léonard, and A. Courville, “Estimating or propagating gradients through stochastic neurons for conditional computation,” *arXiv preprint arXiv:1308.3432*, 2013.

[11] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of The 32nd International Conference on Machine Learning*, 2015, pp. 448–456.

[12] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar *et al.*, “Haswell: The fourth-generation intel core processor,” *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[13] K. Goto and R. A. Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[14] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, “Analytical modeling is enough for high-performance blis,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, p. 12, 2016.

[15] R. Collobert, K. Kavukcuoglu, and C. Farabet, “Torch7: A matlab-like environment for machine learning,” in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[16] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz *et al.*, “The kaldi speech recognition toolkit,” in *IEEE 2011 workshop on automatic speech recognition and understanding*, no. EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

[17] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.