Boosting Deep Neural Network Efficiency with Dual-Module Inference

Liu Liu [1 2] **Lei Deng** [2] **Zhaodong Chen** [2] **Yuke Wang** [1] **Shuangchen Li** [3] **Jingwei Zhang** [3] **Yihua Yang** [3] **Zhenyu Gu** [3] **Yufei Ding** [1] **Yuan Xie** [2]

Abstract

Using deep neural networks (DNNs) in machine learning tasks is promising in delivering high-quality results but challenging to meet stringent latency requirements and energy constraints because of the memory-bound and the compute-bound execution pattern of DNNs. We propose a *big-little* dual-module inference to dynamically skip unnecessary memory accesses and computations to accelerate DNN inference. Leveraging the noise-resilient feature of nonlinear activation functions, we propose to use a lightweight *little* module that approximates the original DNN layer, termed as the *big* module, to compute activations of the insensitive region that are more noise-resilient. Hence, the expensive memory accesses and computations of the *big* module can be reduced as the results are only calculated in the sensitive region. For memory-bound models such as recurrent neural networks (RNNs), our method can reduce the overall memory accesses by 40% on average and achieve 1.54x to 1.75x speedup on a commodity CPU-based server platform with a negligible impact on model quality. In addition, our method can reduce the operations of the compute-bound models such as convolutional neural networks (CNNs) by 3.02x, with only a 0.5% accuracy drop.

1. Introduction

Deep neural networks (DNNs) play a critical role in many areas like image classification (He et al., 2015; Giacinto & Roli, 2001; Zheng et al., 2019), natural language processing (NLP) (Bahdanau et al., 2014; Wu et al., 2016; Graves et al., 2013; He et al., 2019; Wang et al., 2017), and graph processing (Kipf & Welling, 2016; Duvenaud et al., 2015; Klicpera et al., 2018; Yan et al., 2020). While the DNN models are usually trained in data-centers, the pre-trained models can be deployed in both data-centers for cloud-based service and edge devices. During inference, there are primary concerns including latency and energy consumption: low latency is critical for real-time interaction, and low energy can help companies reduce cost in data-centers and increase the endurance of edge devices.

In recent years, extensive studies have shown that quantization and pruning are two effective ways to reduce latency and energy consumption of DNNs (Han et al., 2015b;a; Narang et al., 2017; Zhu & Gupta, 2017; Dai et al., 2018; McKinstry et al., 2018). However, most existing studies apply pruned and quantized models to compute all the output activations; the overall pruning or quantization ratio is limited by the accuracy loss.

In this paper, we observe that most popular activation functions like $ReLU$ in convolutional neural networks (CNNs) and $sigmoid$, $tanh$ in recurrent neural networks (RNNs) demonstrate noise resilience in particular regions, i.e., the negative region of $ReLU$ and the saturation regions of $sigmoid$, $tanh$. We theoretically and experimentally provide evidence that the noise caused by pruning and quantization in these regions is less influential. This observation motivates us that more aggressive pruning and quantization can be applied to these insensitive regions.

Leveraging the noise resilience of DNNs, we propose a *big-little* dual-module inference (DMI) algorithm that regards the original pre-trained module as a *big* module, and use a *little* module that has fewer parameters and lower bit-width to approximate the results of the *big* module in the insensitive regions. Executing one DNN layer at inference time takes the following steps: (1) quantize the input activations and forward them through the *little* module; (2) predict which output neurons belong to the sensitive region based on results from the *little* module; (3) compute the output activations of the *big* module in the predicted sensitive region; (4) combine the outputs of the *little* module in the insensitive region and the outputs of the *big* module with in the sensitive region.

[1]Department of Computer Science, University of California, Santa Barbara [2]Department of Electrical and Computer Engineering, University of California, Santa Barbara [3]Alibaba Group. Correspondence to: Liu Liu <liu_liu@ucsb.edu>, Lei Deng <leideng@ucsb.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

Table 1. Comparison of adding Gaussian noises to the sensitive or insensitive region of LSTM gates.

Case	Cosine similarity before gates				Cosine similarity after gates				PPL
	input	forget	cell	output	input	forget	cell	output	
Sensitive	0.953	0.859	0.952	0.932	0.934	0.946	0.882	0.940	85.70
Insensitive	0.944	0.929	0.943	0.947	0.968	0.987	0.969	0.977	81.79

The weight matrix of the *little* module is constructed as follows. Assuming the weight matrix of *big* model $\boldsymbol{W}^{HH}$ is an $n \times d$ dense matrix, we first randomly initialize a smaller $n \times k$ quantized dense matrix $\boldsymbol{W}^{LL}$ where $k \ll d$. Then, we multiply it with a $k \times d$ random projection matrix $\boldsymbol{P}$ so that $\boldsymbol{W}^{LL}\boldsymbol{P}$ and $\boldsymbol{W}^{HH}$ have the same size. Because $\boldsymbol{P}$ is very sparse and its entries are either ± 1 or 0, the projection doesn't influence the bit-width of $\boldsymbol{W}^{LL}$. Therefore, $\boldsymbol{W}^{LL}\boldsymbol{P}$ is a sparse and quantized matrix that has the shape of $\boldsymbol{W}^{HH}$. The weights of the *little* module are trained in a knowledge-distilling way by mimicking the behaviors of the *big* module. Specifically, following Yim et al. (2017), we minimize the Frobenius norm of the difference between the flows of the solution procedure (FSP) matrix of the *big* and *little* modules. To accurately predict which output neurons belong to the insensitive region, we minimize the Kullback-Leibler (KL) divergence between the output distributions of the *big* and *little* modules. Our theoretical analysis reveals that both targets can be achieved by minimizing the reconstruction error, i.e., mean-squared error, between the output feature maps of the *big* and *little* modules.

Our DMI algorithm can be applied to various types of neural networks. To demonstrate this point, we evaluate its performance on CNNs, LSTM, and GRU. For the memory-bound RNNs, with overall memory accesses reduced by 40% on a commodity CPU-based server platform, our method can achieve 1.54x to 1.75x wall-clock time speedup with negligible impact on model quality. In addition, our method can reduce the operations of the compute-bound CNNs by 3.02x, with only a 0.5% accuracy drop.

2. Motivation

In this section, we discuss the noise resilience of popular activation functions in DNNs including $tanh$, $sigmoid$, and $ReLU$. For clarity, we denote the pre-activation and the noise by x and δ, respectively. Generally, an activation function f is resilient to a noise δ when we have $|f(x+\delta) - f(x)| < \theta$, where θ is a threshold.

For $tanh$, when $|x| \gg 0$, we have

$$|tanh(x+\delta) - tanh(x)| \approx 2e^{-2|x|}|\delta|. \quad (1)$$

Similarly, for $sigmoid$, when $|x| \gg 0$ we have

$$|sigmoid(x+\delta) - sigmoid(x)| \approx e^{-|x|}|\delta|. \quad (2)$$

While for $ReLU$, we have

$$|ReLU(x+\delta) - ReLU(x)| \begin{cases} = 0 & x \leq -|\delta| \\ \leq |\delta| & otherwise \end{cases}. \quad (3)$$

We define the sub-domain of f that is resilient to the noise δ as the insensitive region of f. For a given θ, the insensitive regions of the above three activations are listed as follows

$$\begin{cases} tanh: & |x| > \frac{1}{2}\ln\frac{2|\delta|}{\theta} \\ sigmoid: & |x| > \ln\frac{|\delta|}{\theta} \\ ReLU: & x < \theta - |\delta| \end{cases}. \quad (4)$$

While the insensitivity of $ReLU$ is quite straightforward, we demonstrate our conclusions on $sigmoid$ and $tanh$ with a single LSTM layer for language modeling over PTB dataset. The baseline perplexity (PPL) is 80.64. For each gate, we consider two cases: adding Gaussian noise to the pre-activations before passing through the gate in the sensitive region; in contrast, adding Gassian noise to the insensitive region. We separate the (in)sensitive regions by 50% based on the magnitude of activations.

As listed in Table 1, we report the PPL on the testing set and the average cosine similarity between the activations of the baseline model and the noise-introduced model. Before applying the nonlinear activation functions, the cosine similarity of two cases – adding noise in the sensitive region or the insensitive region – are in the same level. However, we observe that after the nonlinear gates, the cosine similarity in the insensitive case is much closer to one (i.e., fewer output errors) than that in the sensitive case. We further compare the PPL of these two cases, and we observe that introducing noise in the insensitive region causes little quality degradation.

The selection of which output neurons should be in the (in)sensitive region is dynamic and input-dependent. Unlike the static weight sparsity that we can prune the ineffectual connections offline in advance, the dynamic region speculation requires a very lightweight criterion for real-time processing. Taking all these into account, we propose a dual-model inference (DMI) algorithm that efficiently determines (in)sensitive region and significantly saves the memory accesses and computations.

3. Approach

First, we explain the DMI algorithm by taking a fully-connected (FC) layer as an example and then extend it to LSTM, GRU, and CNN. For an FC layer with batch size of one, the operation is typically formulated as $\boldsymbol{z} = \varphi(\boldsymbol{y}), \boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{W}$ is a weight matrix ($\boldsymbol{W} \in \mathbb{R}^{n \times d}$), $\boldsymbol{x}$ is an input vector ($\boldsymbol{x} \in \mathbb{R}^d$), $\boldsymbol{b}$ is a bias vector ($\boldsymbol{b} \in \mathbb{R}^n$), $\boldsymbol{y}$ is a pre-activated output vector ($\boldsymbol{y} \in \mathbb{R}^n$), $\boldsymbol{z}$ is an activated output vector ($\boldsymbol{z} \in \mathbb{R}^n$), and φ is an activation function.

3.1. Overview of Dual-module Philosophy

We have shown in Section 2 that not all values in $\boldsymbol{z}$ need accurate computation, and those belonging to the insensitive region can afford some extent of approximation. In other words, we only need accurate computations and expensive memory accesses in the sensitive region of $\boldsymbol{y}$ and can skip the ones in the insensitive region. With that, we still need approximated results in the insensitive region. Therefore, we propose to learn a lightweight *little* module from the original trained layer, here we refer the original layer as the *big* module. Essentially, our *little* module has low-volume parameters and low bit-width, thus termed as LL module; by contrast, the original *big* module has high-volume parameters and high precision is called HH module. Let the outputs from these two modules be $\boldsymbol{y}^{LL}$ and $\boldsymbol{y}^{HH}$, respectively. If the LL module approximates the HH module well, the final output vector – a mixture of results from the HH and the LL modules – can be assembled by

$$\boldsymbol{y} = \boldsymbol{y}^{HH} \odot \boldsymbol{m} + \boldsymbol{y}^{LL} \odot (1 - \boldsymbol{m}) \tag{5}$$

where $\boldsymbol{m} \in \{0, 1\}^n$ is a binary mask vector for the output switching. m_i equals 1 in the sensitive region while it switches to 0 in the insensitive region. The overall saving comes from skipping memory accesses and computations of the *big* module while paying smaller overhead in accessing and computing the *little* module.

Applying dual-module inference introduces two challenges: first, how to efficiently construct the LL module; second, how to predict which output neurons belong to the (in)sensitive regions.

3.2. Construct the LL Module

As the HH module is the original pre-trained layer, we only need to construct an extra LL module. Delivering a lightweight *little* module at inference time is crucial to achieving real wall-clock time speedup. We emphasize two objectives when designing the LL module: firstly, achieving much lower overhead in terms of computation and memory access than the HH module; secondly, approximating the outputs of HH module accurately.

Lightweight Linear Transformation. We exploit two ways to construct a lightweight approximation of $\boldsymbol{W}^{HH}\boldsymbol{x}$: making $\boldsymbol{W}^{HH}$ sparse and using low bit-width data.

Inspired by random projection, a common technique for dimension reduction while preserving the distances in Euclidean space (Achlioptas, 2003; Bingham & Mannila, 2001; Li et al., 2006; Liu et al., 2019), we first initialize a smaller dense matrix $\boldsymbol{W}^{LL} \in \mathbb{R}^{n \times k}$ where $k < d$, and then transform it by multiplying it with a $k \times d$ sparse random projection matrix $\boldsymbol{P}$ in which

$$P_{ij} = \sqrt{\frac{3}{k}} \times \begin{cases} +1 & with\ probability\ 1/6 \\ 0 & with\ probability\ 2/3 \\ -1 & with\ probability\ 1/6 \end{cases}. \tag{6}$$

In other words, we replace $\boldsymbol{W}^{HH}\boldsymbol{x}$ with $\boldsymbol{W}^{LL}\boldsymbol{P}\boldsymbol{x}$. The original layer takes $O(n \times d)$ MACs (multiply-and-accumulate operations), while our sparse kernel only needs $O(\frac{1}{3} \times k \times d + n \times k)$ MACs. Therefore, using a smaller k can greatly reduce the memory and compute costs.

However, there still should be a lower bound of k (i.e. $\inf(k)$) to maintain the approximation accuracy. We make a hypothesis that the minimum number of parameters in $\boldsymbol{W}^{LL}$ is approximate to the minimum number of parameters in $\boldsymbol{W}^{HH}\boldsymbol{P}^T$ that preserves the Euclidean distance between $\boldsymbol{W}^{HH}$'s row vectors. The intuition behind is that while each row vector in $\boldsymbol{W}^{HH}$ defines the linear transformation of each output channel, once the Euclidean distance is not preserved, there might be fewer effectual channels, which hurts the accuracy.

According to the Theorem 1.1 in Achlioptas (2003), given constant β and ϵ, as long as k satisfies

$$k > k_0 = \frac{4 + 2\beta}{\epsilon^2/2 - \epsilon^3/3} log(n) \tag{7}$$

with probability at least $1 - n^{-\beta}$, for all row vectors $\boldsymbol{u}, \boldsymbol{v} \in Row(\boldsymbol{W}^{HH})$, we have

$$(1-\epsilon)||\boldsymbol{u}^T - \boldsymbol{v}^T||_2^2 \leq ||\boldsymbol{u}\boldsymbol{P}^T - \boldsymbol{v}\boldsymbol{P}^T||_2^2 \leq (1+\epsilon)||\boldsymbol{u}^T - \boldsymbol{v}^T||_2^2. \tag{8}$$

The optimal ϵ and β can be obtained experimentally on validation set. As increasing β is somehow equivalent with decreasing ϵ, we simplify Equation (7) as follows

$$k = \frac{4}{\epsilon^2/2 - \epsilon^3/3} log(n). \tag{9}$$

To further reduce the complexity, we also apply the quantization technique to reduce the bit-wdith of parameters. Specifically, we apply a one-time uniform quantization on $\boldsymbol{W}^{LL}$ and $\boldsymbol{b}^{LL}$ to avoid complicated calculations. Although some other accurate quantization methods are available as well, we find that one-time quantization works well in our

DMI. Besides, the input $\boldsymbol{x}$ is also quantized to $\boldsymbol{x}_Q$ during run-time to reduce the compute cost.

Knowledge Distillation. In order to train the LL module such that it is a good approximation of the HH module, we utilize the Knowledge Distillation method by taking the HH module as the teacher network and the LL module as a student network. According to Yim et al. (2017), transferring the distilled knowledge as the flow of the solution procedure (FSP) matrix between two layers can increase the convergence rate and get better performance.

Let the outputs of two layers be $\boldsymbol{x}$ and $\boldsymbol{z}$. In Yim et al. (2017), the FSP matrix represented by $\boldsymbol{G} = \boldsymbol{x}\boldsymbol{y}^{\boldsymbol{T}}$. In order to transfer the knowledge, we want the FSP matrix of the student network $\boldsymbol{G_s}$ to approximate to the FSP matrix of teach network $\boldsymbol{G_t}$. The loss function is defined as

$$L = ||\boldsymbol{G_t} - \boldsymbol{G_s}||_F^2. \tag{10}$$

In our dual module, we have

$$\begin{aligned} &\boldsymbol{G_t} = \boldsymbol{x}\left(\boldsymbol{y}^{HH}\right)^T,\ \ \boldsymbol{G_s} = \boldsymbol{x}\left(\boldsymbol{y}^{LL}\right)^T, \\ &||\boldsymbol{G_t} - \boldsymbol{G_s}||_F^2 = Tr\left(\boldsymbol{x}\boldsymbol{x}^T\right)||\boldsymbol{y}^{HH} - \boldsymbol{y}^{LL}||_2^2. \end{aligned} \tag{11}$$

As a result, during fine-tuning, the parameters of the HH module (i.e., $\boldsymbol{W}^{HH}$ and $\boldsymbol{b}^{HH}$) are kept frozen while the parameters of the LL module (i.e., $\boldsymbol{W}^{LL}$ and $\boldsymbol{b}^{LL}$) are updated by stochastic gradient descent (SGD) to minimize the following loss function:

$$\begin{aligned} &L = \frac{1}{S}\sum_s ||\boldsymbol{y}^{HH} - \boldsymbol{y}^{LL}||_2^2 = \\ &\frac{1}{S}\sum_s ||(\boldsymbol{W}^{HH}\boldsymbol{x} + \boldsymbol{b}^{HH}) - (\boldsymbol{W}^{LL}\boldsymbol{P}\boldsymbol{x} + \boldsymbol{b}^{LL})||_2^2, \end{aligned} \tag{12}$$

where S is the mini-batch size.

Insensitive Region Prediction. Whether a pre-activation belongs to the insensitive region is predicted based on whether y_i^{LL} is in the insensitive region. Without loss of generality, we assume that $\forall i, y_i^{HH}$ follows some distribution p_{HH} with a probability density function $p_{HH}(x)$. As the $\boldsymbol{y}^{LL}$ is the output of an FC layer, according to the central limit theorem, we can assume that each of its entries follow Gaussian distribution as follows

$$p_{LL} = N(y_\mu^{LL},\ \sigma_{LL}^2) \tag{13}$$

where σ_{LL} is a constant value and y_μ^{LL} gives the prediction of the mean. Similarly, the probability density function is $p_{LL}(x)$.

The difference between these two distributions can be measured by their Kullback-Leibler (KL) divergence, i.e.

$$\begin{aligned} D_{KL}(p_{HH}||p_{LL}) &= \int_{-\infty}^{\infty} p_{HH}(x) ln\left(\frac{p_{HH}(x)}{p_{LL}(x)}\right) dx \\ &= \mathbb{E}_{x\sim p_{HH}}\left[ln\left(p_{HH}(x)\right) - ln\left(p_{LL}(x)\right)\right] \end{aligned} \tag{14}$$

where $\mathbb{E}$ represents the expectation. To make an accurate estimation, we can minimize the above KL divergence, which is equivalent to maximizing $\mathbb{E}_{x\sim p_{HH}}\left[ln(p_{LL}(x))\right]$, and we have

$$\begin{aligned} &\mathbb{E}_{x\sim p_{HH}}\left[ln(p_{LL}(x))\right] = \mathbb{E}\left[ln\left(\frac{1}{\sigma_{LL}\sqrt{2\pi}} e^{-\frac{(y^{HH} - y_\mu^{LL})^2}{2\sigma_{LL}^2}}\right)\right] \\ &\approx -ln(\sigma_{LL}) - \frac{1}{2}ln(2\pi) - \frac{1}{2\sigma_{LL}^2}\mathbb{E}\left[\left(y^{HH} - y_\mu^{LL}\right)^2\right]. \end{aligned} \tag{15}$$

Because of

$$\mathbb{E}\left[\left(y^{HH} - y_\mu^{LL}\right)^2\right] \approx \frac{1}{S}\sum_{i=1}^{S}\left(y_i^{HH} - y_i^{LL}\right)^2 = ||\boldsymbol{y}^{HH} - \boldsymbol{y}^{LL}||_2^2, \tag{16}$$

$D_{KL}(p_{HH}||p_{LL})$ can be equivalently minimized by minimizing $||\boldsymbol{y^{HH}} - \boldsymbol{y^{LL}}||_2^2$ that is just the loss function used in training the LL model (see Equation (12)).

3.3. Determine the Insensitive Region

Given $\boldsymbol{y}^{LL}$, the binary mask $\boldsymbol{m}$ in Equation (5) is generated by predicting which output neurons belong to the insensitive region. Specifically, based on Section 2, we have

$$\begin{cases} sigmoid/tanh: & if\ |y_i^{LL}| > \theta_{th},\ m_i = 0;\ else\ m_i = 1 \\ ReLU: & if\ y_i^{LL} < \theta_{th},\ m_i = 0;\ else\ m_i = 1 \end{cases} \tag{17}$$

where θ_{th} is the threshold can be obtained in the ways as follows.

Fixed Threshold. We can simply assign a constant value to θ_{th} and tune it on validation set.

Adaptive Filter. With a global fixed threshold θ like the fixed threshold, we can adaptively assign a threshold to each layer based on Equation (4), i.e:

$$\theta_{th} = \begin{cases} \frac{1}{2}\ln\frac{2|\delta|}{\theta} & for\ tanh \\ \ln\frac{|\delta|}{\theta} & for\ sigmoid \\ \theta - |\delta| & for\ ReLU \end{cases}. \tag{18}$$

$|\delta|$ is approximated by $\frac{1}{n}||\boldsymbol{y}^{HH} - \boldsymbol{y}^{LL}||_2$, where n is the length of $\boldsymbol{y}^{LL}$. Compared with fixed threshold, the adaptive filter can allow more aggressive thresholds.

Top-K Mask. We can also specifically control the acceleration ratio by taking exactly K elements out of output

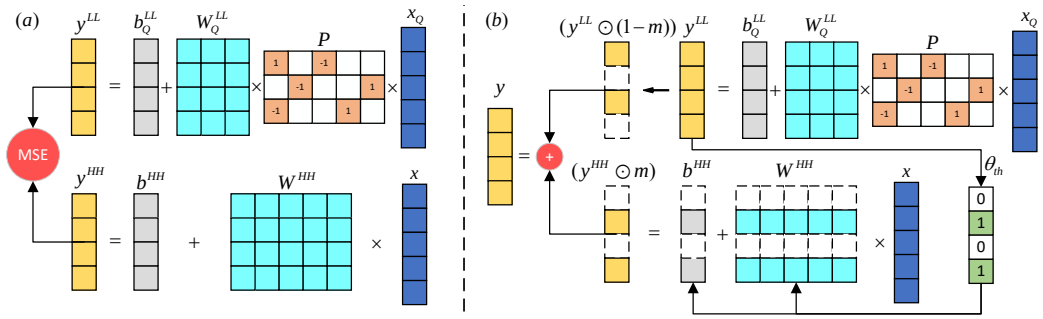

Figure 1. Overview of the Dual-Module Algorithm. (a) Fine-tuning: the LL module is trained with the loss function in Equation (12). (b) Inference: first, the quantized input activation $\boldsymbol{x}_Q$ is injected into the LL module to generate $\boldsymbol{y}^{LL}$; second, mask $\boldsymbol{m}$ is obtained based on $\boldsymbol{y}^{LL}$ and the threshold θ_{th}; third, based on $\boldsymbol{m}$, the elements in $\boldsymbol{y}^{HH}$ are selectively computed to produce $\boldsymbol{y}^{HH} \odot \boldsymbol{m}$; at last, y is obtained using Equation (5).

Algorithm 1 Dual-Module Fine-tuning Algorithm

Data: HH module parameters $\boldsymbol{W}^{HH}$, $\boldsymbol{b}^{HH}$; random projection matrix $\boldsymbol{P}$; input batch $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_S]$.
Result: quantized LL module parameters: $\boldsymbol{W}_Q^{LL}$ and $\boldsymbol{b}_Q^{LL}$.

for $it \in$ all iterations **do**
  $\boldsymbol{X}_Q = Q(\boldsymbol{X})$;
  $[\boldsymbol{y}_1^{LL}, ..., \boldsymbol{y}_S^{LL}] = \boldsymbol{W}_Q^{LL}\boldsymbol{P}\boldsymbol{X}_Q + \boldsymbol{b}_Q^{LL}$;
  $[\boldsymbol{y}_1^{HH}, ..., \boldsymbol{y}_S^{HH}] = \boldsymbol{W}^{HH}\boldsymbol{X} + \boldsymbol{b}^{HH}$;
  $L_{MSE} = \frac{1}{S}\sum_s ||\boldsymbol{y}_s^{HH} - \boldsymbol{y}_s^{LL}||_2^2$;
  update $\boldsymbol{W}_Q^{LL}, \boldsymbol{b}_Q^{LL}$ with $SGD(\min L_{MSE})$;
end

neurons, where K can be a hyper-parameter tuned on validation set. Intuitively, these K output neurons should be taken from the sensitive region, so we have

$$\theta_{th} = \begin{cases} top_K(|\boldsymbol{y}^{LL}|) & for\ tanh/sigmoid \\ top_K(\boldsymbol{y}^{LL}) & for\ ReLU \end{cases}. \quad (19)$$

We introduce an insensitive ratio as the number of outputs using the results of the *little* module over the entire outputs. The ratio can be interpreted as the zero ratio in the binary mask $\boldsymbol{m}$. The higher insensitive ratio will have fewer computations and memory accesses in the *big* module. The choice of an accurate ratio determines the model inference quality, and it is a knob to trade-off the inference quality vs. latency at run-time.

3.4. Overview of the Dual-Module Algorithm

Here we summarize the overall implementation of our dual-module algorithm during fine-tuning and inference.

Fine-tuning. As illustrated in Figure 1(a), the LL modules in different layers are fine-tuned individually with the loss function in Equation (12). The detailed implementation is in given Algorithm 1.

Algorithm 2 Dual-Module Inference Algorithm

Data: HH module parameters $\boldsymbol{W}^{HH}$, $\boldsymbol{b}^{HH}$; quantized LL module parameters $\boldsymbol{W}_Q^{LL}$, $\boldsymbol{b}_Q^{LL}$; threshold θ_{th} to determine $\boldsymbol{m}$; random projection matrix $\boldsymbol{P}$; current input $\boldsymbol{x}$.
Result: Final output $\boldsymbol{y}$

(1) $\boldsymbol{x}_Q = Q(\boldsymbol{x})$;
(2) $\boldsymbol{y}^{LL} = \boldsymbol{W}_Q^{LL}\boldsymbol{P}\boldsymbol{x}_Q + \boldsymbol{b}_Q^{LL}$;
(3) Generating $\boldsymbol{m}$ according to Section 3.3;
(4-5) **foreach** $m_i \in \boldsymbol{m}$ **do**
  if $m_i == 1$ **then** $y_i = y_i^{HH} = \varphi(\boldsymbol{W}^{HH}[i,:]\boldsymbol{x} + b_i^{HH})$;
  else $y_i = y_i^{LL}$;
end

Inference. The dual-module inference (DMI) is illustrated in Figure 1(b) based on Algorithm 2. After obtaining fine-tuned $\boldsymbol{W}_Q^{LL}$ and $\boldsymbol{b}_Q^{LL}$, dual-module inference takes the following steps: (1) quantize input $\boldsymbol{x}$ with $\boldsymbol{x}_Q = Q(\boldsymbol{x})$, where $Q(\cdot)$ is a quantization function; (2) obtain the approximated output $\boldsymbol{y}^{LL}$ by performing $\boldsymbol{y}^{LL} = \boldsymbol{W}_Q^{LL}\boldsymbol{P}\boldsymbol{x}_Q + \boldsymbol{b}_Q^{LL}$; (3) generate the binary mask $\boldsymbol{m}$ according to Section 3.3; (4) calculate the elements y_i^{HH} $s.t.$ $m_i = 1$ with $y_i^{HH} = \boldsymbol{W}^{HH}[i,:]\boldsymbol{x} + b_i^{HH}$; (5) produce the final output $\boldsymbol{y}$ according to the assembling in Equation (5).

3.5. Apply to Various Types of Neural Networks

The above example on FC layer can easily generalize to various types of neural networks. Here we use LSTM and CNNs as examples.

Recurrent Neural Networks. There are two major differences between an LSTM layer and an FC layer: (1) the computation of each gate involves two GEMV operations; (2) there is an additional temporal dimension in LSTM. For the former, we apply the lightweight linear transformation in Section 3.2 to both GEMVs. For the latter, we modify the

loss function to guarantee the approximation performance of the LL module at all timesteps. Taking the forget gate as an example, the loss function L_{MSE} in Algorithm 1 is modified to

$$L_{MSE} = \frac{1}{ST}\sum_{s}\sum_{t}||\boldsymbol{y}_f^{HH}(t) - \boldsymbol{y}_f^{LL}(t)||_2^2,$$
$$\boldsymbol{y}_f^{LL}(t) = \boldsymbol{b}_{fQ}^{LL} + \boldsymbol{W}_{fxQ}^{LL}\boldsymbol{P}_{\boldsymbol{x}}\boldsymbol{x}_Q(t) + \boldsymbol{W}_{fhQ}^{LL}\boldsymbol{P}_{\boldsymbol{h}}\boldsymbol{h}_Q(t-1),$$
$$\boldsymbol{y}_f^{HH}(t) = \boldsymbol{b}_f + \boldsymbol{W}_{fx}\boldsymbol{x}(t) + \boldsymbol{W}_{fh}\boldsymbol{h}(t-1). \tag{20}$$

Convolutional Neural Networks. For a CONV layer, We can apply dual-module algorithm to CNN by first doing the *im2col* transformation on input tensor (Chetlur et al., 2014). Then, the input and output become matrices rather than vectors, but the overall algorithm is the same as in Section 3.4.

Batch Normalization. Batch normalization (BN) (Ioffe & Szegedy, 2015) is widely applied in DNNs. During inference, BN normalizes the input activations with

$$\hat{\boldsymbol{x}} = \gamma(\frac{\boldsymbol{x}-\mu}{\sigma}) + \beta, \tag{21}$$

where γ, β are trainable parameters, and μ, σ are the moving average of the mean and standard deviation of activations collected during training.

Our dual-module algorithm is compatible with BN. When BN is applied before the activation function (Chen et al., 2020), i.e., $\varphi(BN(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}))$, BN can be merged into the linear transformation as follows

$$\varphi\left(BN\left(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}\right)\right) = \varphi\left(\frac{\gamma}{\sigma}\boldsymbol{W}\boldsymbol{x} + \left(\boldsymbol{b}+\beta-\frac{\gamma}{\sigma}\mu\right)\right). \tag{22}$$

We can have $\hat{\boldsymbol{W}} = \frac{\gamma}{\sigma}\boldsymbol{W}$ and $\hat{\boldsymbol{b}} = \boldsymbol{b} + \beta - \frac{\gamma}{\sigma}\mu$, then our algorithm can be directly applied. When BN is applied after the activation function, our $\varphi\left(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}\right)$ structure is not influenced by BN.

4. Evaluation

Our dual-module method is generally applicable to both RNNs and CNNs to improve the inference efficiency. Here we choose a representative set of RNN and CNN models to demonstrate the effectiveness of our method. In our supplementary material, we have more extensive results as well as evaluation methodology and experimental settings. We use PyTorch to train the *little* module and evaluate the inference quality. We train the *little* module while freezing the parameters of the *big* module, and we use the same training set and validation set to run the SGD optimization.

4.1. Experimental Results on RNNs

As memory access is the bottleneck in RNN-based inference, we apply DMI with the focus of reducing overall memory access while keeping the overhead of executing the *little* module small. As shown in Figure 2, we compare accessed data and operations between the single-module – the baseline case – and the *little* module using a set of LSTM and GRU layers. On average, the *little* module accounts only 8% memory overhead and 35% operation overhead compared with the base. Note that we count the number of operations in Figure 2 regardless of precision; and the computation overhead of the *little* module can be further reduced using a low-precision implementation.

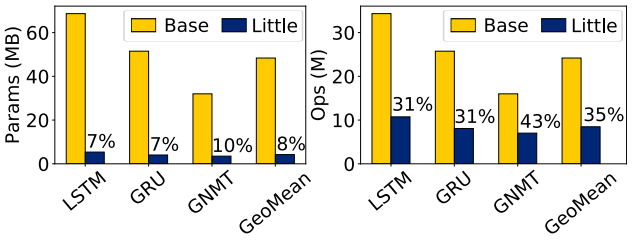

Figure 2. Comparison of the amount of accesses and operations between baseline layers and the *little* module of dual-module enhanced RNN-based models.

Our method is evaluated on CPU-based server platform (Intel(R) Xeon(R) CPU E5-2698 v4) as most inference workloads run on CPUs (Park et al., 2018). The baseline implementation is the PyTorch CPU version with Intel MKL (version 2019.4) as the back-end BLAS kernel library. Our custom implementation uses a multi-threaded MKL dot-product kernel at BLAS level-1 to perform the *big* module instead of BLAS level-2 or level-3 kernels. In our kernel implementation, we do not explicitly generate masks, but directly compare little module results with predetermined thresholds in $OpenMP$ parallel-for loops. The kernel-wise performance is measured as wall-clock time and averaged with 1000 runs, assuming cold cache at the execution of each RNN cell. Accessing weights from off-chip memory at each time-step represents the real-world cases, for example, the decoder execution of sequence-to-sequence modeling.

Language Modeling. Our implementations of LSTMs/GRUs are adapted from the word-level language modeling example from PyTorch using the same hyper-parameters to train baseline models. We report the word-level perplexity (PPL) as the measure of model quality. As listed in Table 2, we vary the insensitive ratio to show the quality-performance trade-off; the larger insensitive ratio indicates more results are from the *little* module and less memory overhead to perform the *big* module. As we increase the insensitive ratio, we observe the degradation of quality as the PPL increases during a gradual reduction in execution time. When the insensitive ratio is 50%, the PPL is slightly increased to 81.36, which is negligible in language modeling tasks, while we can gain 1.67x inference speedup.

Table 2. RNN quality and execution time (ms). Ln means a LSTM layer with h hidden units; G is short for GRU.

Insensitive	LM, L1500				LM, G1500				GNMT, L1024			
Ratio	PPL	Diff.	Time	Speedup	PPL	Diff.	Time	Speedup	BLEU	Diff.	Time	Speedup
Baseline	80.64	n/a	1.477	1.00x	85.48	n/a	1.182	1.00x	24.32	n/a	0.838	1.00x
10%	80.72	-0.08	1.315	1.12x	85.62	-0.14	1.024	1.15x	24.33	0.01	0.679	1.23x
30%	80.56	0.08	1.095	1.35x	86.01	-0.53	0.869	1.36x	24.18	-0.14	0.541	1.55x
50%	81.36	-0.72	0.885	1.67x	88.73	-3.25	0.726	1.63x	23.73	-0.59	0.480	1.75x
70%	87.48	-6.83	0.641	2.30x	98.09	-12.61	0.545	2.17x	21.92	-2.40	0.360	2.33x
90%	109.37	-28.73	0.380	3.89x	122.75	-37.27	0.350	3.38x	11.77	-12.55	0.243	3.45x

We further report the results using single-layer GRUs on word-level language modeling tasks as in Table 2. Using dual-module method on GRUs expresses the similar quality-performance trade-off as on LSTMs.

Neural Machine Translation. Given the promising results on language modeling, we further investigate Neural Machine Translation (NMT), which is a popular end-to-end learning approach for automated translation (Wu et al., 2016) and a standard benchmark model for inference as in MLPerf[1]. Our experiments show the de-tokenized BLEU score to measure the model quality on the public WMT16 English-German dataset. The base model[2] consists of a four-layer stacked LSTM in both the encoder and the decoder of the sequence-to-sequence modeling. We focus on the speedup of the decoder since it is the most memory intensive and the most time-consuming part (about 95%).

We replace the LSTM layers in the decoder with our proposed dual-module LSTM layers. Similar to the single-layer LSTM results, using the *little* module computed results in the insensitive region can reduce the overall memory access while maintaining the model quality. As listed in Table 2, our method can achieve imperceptible BLEU score degradation while accelerating inference by 1.75x. When compromising more translation quality, e.g. decreasing the BLEU score by 2.4, our method can achieve more than 2x speedup.

4.2. Experimental Results on CNNs

Using DMI on CNNs can be regarded as pursuing output sparsity. In Table 3, we compare the classification accuracy and FLOPs reduction of our DMI method with other state-of-the-art methods on predicting $ReLU$-induced output sparsity when ResNet-18 is used for ImageNet classification. The results show that DMI outperforms other methods in delivering better trade-off between the accuracy drop and the FLOPs reduction. With accuracy degrading only 0.5%, our method can achieve 3.02x FLOPs reduction.

Pixel-wise dynamic output sparsity can be accelerated by either customized GEMM kernel (Nisa et al., 2018) or specialized hardware (Akhlaghi et al., 2018). We could translate the FLOPs reduction on CNNs by DMI to practical speedup on either commodity processors or hardware accelerators. Energy efficiency is another important criterion to evaluate any CNN inference execution, especially for mobile and edge devices. Here we show the energy consumption of DMI, normalized to the energy of running the baseline dense layers of each residual block in ResNet-18. The purpose of Figure 3 is to estimate the energy consumption with a focus on the portion of MAC operations which consume the majority of total energy in compute-bound CNNs. The methodology is multiplying the total number of MAC operations of big/little modules with the energy (J) per MAC operation accordingly. The energy/op numbers are from synthesized hardware evaluation. As shown in Figure 3, the energy efficiency improvement of using DMI is from 1.7x to 4.9x; on average, our method can improve the energy efficiency by 2.9x.

Table 3. Comparison of the Top-1 accuracy and FLOPs reduction of our method with prior work on dynamic sparsity. The baseline model is ResNet-18 on ImageNet.

Method	Acc. (%)	Diff. (%)	FLOPs reduction
Dense (*torchvision*)	69.7	n/a	1.00x
LCL (Dong et al., 2017)	66.3	-3.4	1.53x
FBS (Gao et al., 2018)	68.2	-1.5	1.98x
SeerNet (Cao et al., 2019)	69.3	-0.4	1.67x
CGNet (Hua et al., 2019)	68.8	-0.9	1.93x
DMI (Ours)	**69.2**	**-0.5**	**3.02x**

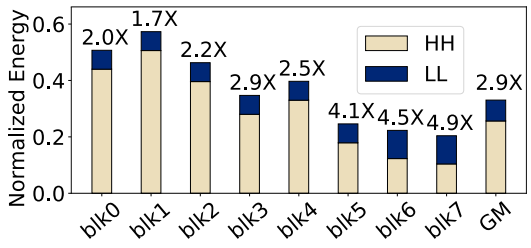

Figure 3. Energy efficiency of each residual block in ResNet-18.

[1]https://mlperf.org/inference-overview/

[2]From https://github.com/NVIDIA/DeepLearningExamples

4.3. Discussion on Dimension Reduction

Dimension reduction is an integral part of our DMI method to reduce the number of parameters of the *little* module. Here, we study the impact of different levels of dimension reduction on the model quality and performance. We conduct experiments on language modeling using a single-layer LSTM of 1500 hidden units. We quantize the *little* module to INT8 and reduce the hidden dimension from 1500 to three different levels, which are calculated by Equation (9). We fix the insensitive ratio at 50% across this set of experiments. As in Table 4, the higher dimension of the *little* module, the better approximation the *little* module can perform. More aggressive dimension reduction can further gain more speedup at the cost of more quality degradation: hidden dimension reduced to 417 and 266 can have 1.67x and 1.71x speedup but increase the PPL by 0.72 and 2.87, respectively.

Table 4. Sensitivity study of dimension reduction.

Dimension	PPL	Speedup	*little*	*big*
1500 (baseline)	80.64	1.00x	0%	100%
966 ($\epsilon = 0.3$)	80.40	1.37x	22%	44%
417 ($\epsilon = 0.5$)	81.36	1.67x	12%	47%
266 ($\epsilon = 0.7$)	83.51	1.71x	8%	46%

We further show the overhead of performing the computation of the *little* module. As listed in the last two columns in Table 4, we measure the execution time of computing the *little* module and the operation-reduced *big* module. The execution time is normalized to the baseline case, i.e., the execution time of the standard LSTM, to highlight the percentage of overheads. When the hidden dimension is reduced to 966, the overhead of the *little* module accounts 22% while the execution time of the *big* module is cut off by half[3]. In our experiments, we choose $\epsilon = 0.5$ as the default parameter as it demonstrates good trade-off between quality and speedup in our study. When further reducing the hidden dimension to 266, there is only a slight improvement on speedup compared with the hidden size of 417 in the *little* module, where the overhead of the *little* module is already small enough, but the quality drop is significant.

4.4. Discussion on Quantization

Weight quantization of the *little* module is another integral part of constructing the *little* module. We show the impact of different quantization levels on the model quality and the parameter size. After training the *little* module, using the same settings as in Section 4.1, we can quantize the weights to lower precision to reduce the memory access on top of the dimension reduction. As listed in Table 5, more aggressive quantization leads to smaller parameter size that can reduce the overhead of computing the *little* module; on the other hand, the approximation of the *little* module is compromised by quantization. We can quantize the *little* module up to INT4 without significant quality degradation. Using lower precision would degrade the quality while decreasing the parameter size. Normally, we choose INT8 as the quantization level since we leverage off-the-shelf INT8 GEMM kernel in MKL. We expect more speedup once the *little* module overhead can be further reduced by leveraging INT4 compute kernels or running on specialized hardware.

Table 5. Inference quality and parameter size comparison under different levels of quantization on the *little* module

Precision	Base	FP32	INT16	INT8	INT4	INT2
PPL	80.64	81.28	81.18	81.36	81.47	82.43
MSE	n/a	0.408	0.425	0.444	0.451	0.68
Params.	68.7	19.1	9.6	4.8	2.4	1.2

5. Related Work

Compressing DNN models via data quantization, weight sparsity, and knowledge distillation is promising to deliver efficient deployment for inference. Quatization methods on weights and activations have been proposed to reduce model size and operation precision (Zhu et al., 2016; Xu et al., 2018; Wang et al., 2018; Choi et al., 2018). Weight pruning has been proposed to reduce the parameters of a pre-trained model (Han et al., 2015b;a). While fine-grained pruning could reduce the number of parameters (Narang et al., 2017; Zhu & Gupta, 2017; Dai et al., 2018), indexing irregular non-zero weights causes extra memory cost and would offset the benefits from reducing parameter size; it is hard to gain practical acceleration on general-purpose hardware or need hardware specialization (Mao et al., 2017). Although structural pruning (Wen et al., 2017) and knowledge distillation (Polino et al., 2018) could achieve speedup, the applicability on more complicated tasks such as NMT on large-scale datasets is unstudied; besides, those methods require extensive and iterative retraining via regularization that would increase the training cost and difficult to find a solution.

Other than model compression techniques, many studies propose to skip computations dynamically based on certain criterion such as layer-wise early exit (Bolukbasi et al., 2017) and $ReLU$-induced sparsity prediction in CNNs (Dong et al., 2017; Gao et al., 2018; Cao et al., 2019; Hua et al., 2019). Sparsity prediction is only a special case of our dual-module inference method. The results of the *little* module are used in the insensitive region instead of only for prediction and then discarded. The special cell structure and the temporal input similarity have enabled computation and up-

[3]We measured the execution time with multi-threading.

date skipping in RNNs (Neil et al., 2017; Zhang et al., 2018; Campos et al., 2018). However, those methods depend on certain applications and lack of evaluation on NLP tasks such as language modeling and machine translation.

We are the first that proposes a general and principled method to reduce memory accesses and computations of DNNs, with general applicability to LSTMs, GRUs, and CNNs. Our method, by no means, is supposed to replace model compression but provide an orthogonal approach to accelerate DNN inference without compromising the model expressive power. Using the analogy of knowledge distillation, we do not simply deploy a student network learned from the teacher network. Instead, we let the teacher network help with the student – the *little* module learned from the base module – and collaboratively perform inference with reduced memory accesses and computations.

6. Conclusion

In this paper, we describe a *big-little* dual-module inference method to boost the execution efficiency of DNNs. We leverage the noise resilience of nonlinear activation functions by using the lightweight *little* module to compute for the insensitive region and using the *big* module with skipped memory access and computation to compute for the sensitive region. Our method can reduce overall memory access by near half for the memory-bound RNNs and achieve 1.54x to 1.75x wall-clock time speedup without significant degradation on model quality. For the compute-bound CNNs, our method can achieve 3.02x operation reduction with only a 0.5% accuracy drop.

Acknowledgment

We thank all the anonymous reviewers for their helpful suggestions. This material is based upon work supported by the National Science Foundations (NSF) under Grant No. 1719160, 1725447, 1730309, and 1925717.

References

Achlioptas, D. Database-friendly random projections: Johnson-lindenstrauss with binary coins. Journal of computer and System Sciences, 66(4):671–687, 2003.

Akhlaghi, V., Yazdanbakhsh, A., Samadi, K., Gupta, R. K., and Esmaeilzadeh, H. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp. 662–673. IEEE, 2018.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.

Bingham, E. and Mannila, H. Random projection in dimensionality reduction: applications to image and text data. In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 245–250. ACM, 2001.

Bolukbasi, T., Wang, J., Dekel, O., and Saligrama, V. Adaptive neural networks for efficient inference. In Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML'17, pp. 527–536. JMLR.org, 2017. URL http://dl.acm.org/citation.cfm?id=3305381.3305436.

Campos, V., Jou, B., i Nieto, X. G., Torres, J., and Chang, S.-F. Skip RNN: Learning to skip state updates in recurrent neural networks. In International Conference on Learning Representations, 2018. URL https://openreview.net/forum?id=HkwVAXyCW.

Cao, S., Ma, L., Xiao, W., Zhang, C., Liu, Y., Zhang, L., Nie, L., and Yang, Z. Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 11216–11225, 2019.

Chen, Z., Deng, L., Wang, B., Li, G., and Xie, Y. A comprehensive and modularized statistical framework for gradient norm equality in deep neural networks. IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1–1, 2020.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759, 2014.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., and Gopalakrishnan, K. Pact: Parameterized clipping activation for quantized neural networks. arXiv preprint arXiv:1805.06085, 2018.

Dai, X., Yin, H., and Jha, N. K. Grow and prune compact, fast, and accurate lstms. arXiv preprint arXiv:1805.11797, 2018.

Dong, X., Huang, J., Yang, Y., and Yan, S. More is less: A more complicated network with less inference complexity, 2017.

Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. Convolutional networks on graphs for learning molecular fingerprints. In Advances in neural information processing systems, pp. 2224–2232, 2015.

Gao, X., Zhao, Y., Łukasz Dudziak, Mullins, R., and zhong Xu, C. Dynamic channel pruning: Feature boosting and suppression, 2018.

Giacinto, G. and Roli, F. Design of effective neural network ensembles for image classification purposes. Image and Vision Computing, 19(9-10):699–707, 2001.

Graves, A., Mohamed, A.-r., and Hinton, G. Speech recognition with deep recurrent neural networks. In 2013 IEEE international conference on acoustics, speech and signal processing, pp. 6645–6649. IEEE, 2013.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149, 2015a.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In Advances in neural information processing systems, pp. 1135–1143, 2015b.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision, pp. 1026–1034, 2015.

He, Y., Sainath, T. N., Prabhavalkar, R., McGraw, I., Alvarez, R., Zhao, D., Rybach, D., Kannan, A., Wu, Y., Pang, R., et al. Streaming end-to-end speech recognition for mobile devices. In ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6381–6385. IEEE, 2019.

Hua, W., Zhou, Y., De Sa, C. M., Zhang, Z., and Suh, G. E. Channel gating neural networks. In Advances in Neural Information Processing Systems, pp. 1884–1894, 2019.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.

Klicpera, J., Bojchevski, A., and Günnemann, S. Predict then propagate: Graph neural networks meet personalized pagerank. arXiv preprint arXiv:1810.05997, 2018.

Li, P., Hastie, T. J., and Church, K. W. Very sparse random projections. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 287–296. ACM, 2006.

Liu, L., Deng, L., Hu, X., Zhu, M., Li, G., Ding, Y., and Xie, Y. Dynamic sparse graph for efficient deep learning. In International Conference on Learning Representations, 2019. URL https://openreview.net/forum?id=H1goBoR9F7.

Mao, H., Han, S., Pool, J., Li, W., Liu, X., Wang, Y., and Dally, W. J. Exploring the regularity of sparse structure in convolutional neural networks. arXiv preprint arXiv:1705.08922, 2017.

McKinstry, J. L., Esser, S. K., Appuswamy, R., Bablani, D., Arthur, J. V., Yildiz, I. B., and Modha, D. S. Discovering low-precision networks close to full-precision networks for efficient embedded inference. arXiv preprint arXiv:1809.04191, 2018.

Narang, S., Elsen, E., Diamos, G., and Sengupta, S. Exploring sparsity in recurrent neural networks. arXiv preprint arXiv:1704.05119, 2017.

Neil, D., Lee, J. H., Delbruck, T., and Liu, S.-C. Delta networks for optimized recurrent network computation. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 2584–2593. JMLR. org, 2017.

Nisa, I., Sukumaran-Rajam, A., Kurt, S. E., Hong, C., and Sadayappan, P. Sampled dense matrix multiplication for high-performance machine learning. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pp. 32–41. IEEE, 2018.

Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., Law, J., Malani, P., Malevich, A., Nadathur, S., et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. arXiv preprint arXiv:1811.09886, 2018.

Polino, A., Pascanu, R., and Alistarh, D. Model compression via distillation and quantization. In International Conference on Learning Representations, 2018. URL https://openreview.net/forum?id=S1XolQbRW.

Wang, P., Xie, X., Deng, L., Li, G., Wang, D., and Xie, Y. Hitnet: hybrid ternary recurrent neural network. In Advances in Neural Information Processing Systems, pp. 604–614, 2018.

Wang, Y., Skerry-Ryan, R., Stanton, D., Wu, Y., Weiss, R. J., Jaitly, N., Yang, Z., Xiao, Y., Chen, Z., Bengio, S., et al. Tacotron: Towards end-to-end speech synthesis. arXiv preprint arXiv:1703.10135, 2017.

Wen, W., He, Y., Rajbhandari, S., Zhang, M., Wang, W., Liu, F., Hu, B., Chen, Y., and Li, H. Learning intrinsic sparse structures within long short-term memory, 2017.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144, 2016.

Xu, C., Yao, J., Lin, Z., Ou, W., Cao, Y., Wang, Z., and Zha, H. Alternating multi-bit quantization for recurrent neural networks. arXiv preprint arXiv:1802.00150, 2018.

Yan, M., Chen, Z., Deng, L., Ye, X., Zhang, Z., Fan, D., and Xie, Y. Characterizing and understanding gcns on gpu. arXiv preprint arXiv:2001.10160, 2020.

Yim, J., Joo, D., Bae, J., and Kim, J. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4133–4141, 2017.

Zhang, X., Xie, C., Wang, J., Zhang, W., and Fu, X. Towards memory friendly long-short term memory networks (lstms) on mobile gpus. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 162–174, Oct 2018. doi: 10.1109/MICRO.2018.00022.

Zheng, W., Chen, Z., Lu, J., and Zhou, J. Hardness-aware deep metric learning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 72–81, 2019.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. arXiv preprint arXiv:1612.01064, 2016.

Zhu, M. and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv preprint arXiv:1710.01878, 2017.