Characterizing the Workload of Dynamic Neural Networks

Zhenyu Zou, Rui Huang
zzou24@wisc.edu, rhuang94@wisc.edu
University of Wisconsin-Madison

12/17/2019

1 Introduction

The last few years have witnessed tremendous success of Long-Short Term Memory (LSTM) in a wide range of machine learning applications, especially in natural language processing (NLP). Thanks to its unparalleled strengths in processing sequential input, LSTM is widely adopted in text-based tasks, including language modeling [8, 7], named entity recognition [11, 2] and machine translation [1, 12].

The magical power of LSTM arises from its computation unit – LSTM cells. The memory in the cell keeps track of current status and will be constantly updated according to new inputs. The various kinds of gates in the cell decide how much of the current input enters cell memory and how much of the cell memory contributes to the output. Each time the LSTM cell takes one element from the input sequence, updates the cell memory, and generates an output based on both current input and cell memory. This process is iterated until the whole sequence is consumed.

LSTM excels in capturing long-term dependencies due to the nature of its network architecture. However, this network architecture also poses huge challenges to existing deep learning frameworks. Different from convolutional neural networks (CNN) where computation graph is deterministic without branches and loops, LSTM employs loops to iterate over a sequence of input. Therefore, static control flow implemented by existing frameworks will not apply to LSTM any more. In light of this, dynamic control flow has been invented to accommodate dynamic neural networks, such as LSTM [21, 20].

Dynamic control flow is a remarkable invention because it makes large-scale LSTM model training possible. However, the workload features of dynamic control flow vary a lot from static control flow and they have not been well studied by the literature. In this paper, we believe that it is of great importance to characterize the workload features of dynamic neural networks in existing deep learning frameworks, because it will give us valuable insights of how to train LSTM more efficiently. Furthermore, it could also pinpoint some potential flaws of dynamic neural network training in existing frameworks and clarify directions for future improvement.

In this paper, we focus on a particular kind of dynamic neural network, LSTM, and make three hypotheses on the workload features and resource usage in the training process of LSTM models.

Lower parallelism than CNN Since neural networks are usually heavy-size models with a large number of parameters and computations, it is of great importance to use parallel computation to reduce model training time. Convolution, the most important operation in CNN, is essentially a parallel operation. However, due to its serial computation fashion of LSTM, it has much less potential to be computed in parallel. Most deep learning frameworks give users the option of using GPUs for speed-up in model training, because GPU supports large-scale parallel vector computation. This introduces a significant speed-up to CNN models. However, we assume that GPU utilization in LSTM training will not be as high as that in CNN training because of its lower parallelism. In other words, the speed-up from CPU to GPU for LSTM will not be as significant as that for CNN.

Sensitive to model size and data size Because LSTM has lower parallelism and can only be computed in serial with respect to the input data, its training time is largely dependent on input data size (sequence length). However, convolution can be computed in fully parallel on the input data, so CNN's running time is not greatly affected by input size. In the same vein, the performance of LSTM is also more sensitive to the change of model size (such as hidden dimensions and batch size) than that of CNN.

Memory usage and memory bandwidth Because all the temporary status and intermediate values in the LSTM cell need to be stored in memory at each iteration, for computation in next steps and for backpropgation purposes, the memory usage of training LSTM networks can be very high. Because the total amount of memory on GPU is very limited and thus can be used up easily, Tensorflow introduces a memory swapping mechanism to continuously move history tensors from GPU

memory to CPU memory at forward stage, and then from CPU memory to GPU memory at backward stage. This is a very efficient way to avoid the out-of-memory problem, but we hypothesize that the CPU-GPU memory bandwidth will be very high and could become a bottleneck during the training process.

In the following sections, we investigate the workload features of LSTM in detail and carry out extensive experiments to verify the three hypotheses above respectively. We choose image classification as our benchmark task because both CNNs and LSTMs have achieved good performance on it. We use MNIST[1] as our benchmark dataset.

2 Background

With the rapid rising of computation power in modern machines, deep neural networks become widely adopted in a wide range of machine learning applications. Among the numerous set of different kinds of deep neural networks, CNN and LSTM are probably the most popular ones. Before describing how to characterize their workload, we first briefly introduce the computation mechanism of these two models.

2.1 Computation in CNN

CNN is commonly used in image-related applications, including image classification [9, 4], object detection [17, 14], semantic segmentation [15], etc. It has fantastic power to capture local neighborhood information and finally merge it into global information.

CNN is based on a fundamental operation called convolution, where a kernel matrix is applied on an input image to produce a feature map. For each pixel in the input image, we sum up all of its surrounding pixels with the corresponding weights defined in the corresponding positions in the kernel matrix and generate an output pixel. Mathematically,

$$\mathbf{Y}_{r,c} = \sum_{i=-\frac{k}{2}}^{\frac{k}{2}} \sum_{j=-\frac{k}{2}}^{\frac{k}{2}} \mathbf{K}_{i+\frac{k}{2},j+\frac{k}{2}} \mathbf{X}_{r+i,c+j} \tag{1}$$

where $\mathbf{Y}$ is the output feature map, $\mathbf{X}$ is the input image, $\mathbf{K}$ is the kernel matrix, and k is the size of the kernel matrix. Intuitively, we are shifting a convolution window from the top-left to the bottom-right of an input image and compute an output value at each possible window position. Note that this operation is highly parallelable since each output pixel can be computed independently and concurrently. This parallelism can be largely utilized on GPUs for CNN model training.

[1] http://yann.lecun.com/exdb/mnist/

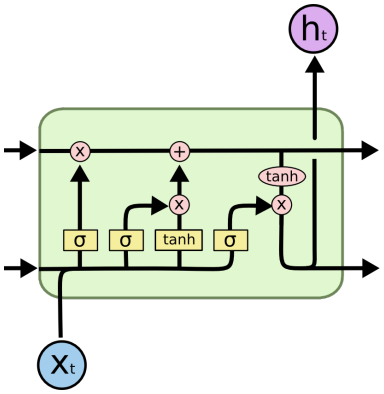

Figure 1: Computation in LSTM cells

2.2 Computation in LSTM

LSTM was invented by Hochreiter & Schmidhuber in 1997 [5] and is now widely adopted in text-based applications, including language modeling [8, 7], named entity recognition [11, 2] and machine translation [1, 12]. LSTM is well known for its capacity of capturing long-term dependency in an input sequence. More specifically, LSTM takes an sequence as input, processes one element from the input sequence at a time, and generates an output based on both current input and cell memory. This process is iterated until the whole sequence is consumed. As illustrated in Figure 1, the computation in the LSTM cell is formulated as:

$$i_t = \sigma(W_{hi} h_{t-1} + W_{xi} x_t + b_i) \tag{2}$$

$$f_t = \sigma(W_{hf} h_{t-1} + W_{xf} x_t + b_f) \tag{3}$$

$$o_t = \sigma(W_{ho} h_{t-1} + W_{xo} x_t + b_o) \tag{4}$$

$$C_t^* = tanh(W_{hC} h_{t-1} + W_{xC} x_t + b_C) \tag{5}$$

$$C_t = \sigma(f_t \odot C_{t-1} + i_t \odot C_t^*) \tag{6}$$

$$h_t = o_t \odot tanh(C_t) \tag{7}$$

i_t is the input gate, deciding how much of the current input enters cell memory. o_t is the output gate, controling how much of the cell memory contributes to the output. f_t is the forget gate, determining how much of the previous cell memory is dropped. The memory in the cell, C_t, keeps track of current status and will be constantly updated given new inputs.

As we can see from the above equations, LSTM's computation process is highly serial because each stage is based on results from previous stages. Therefore, its training time could be largely influenced by input sequence length.

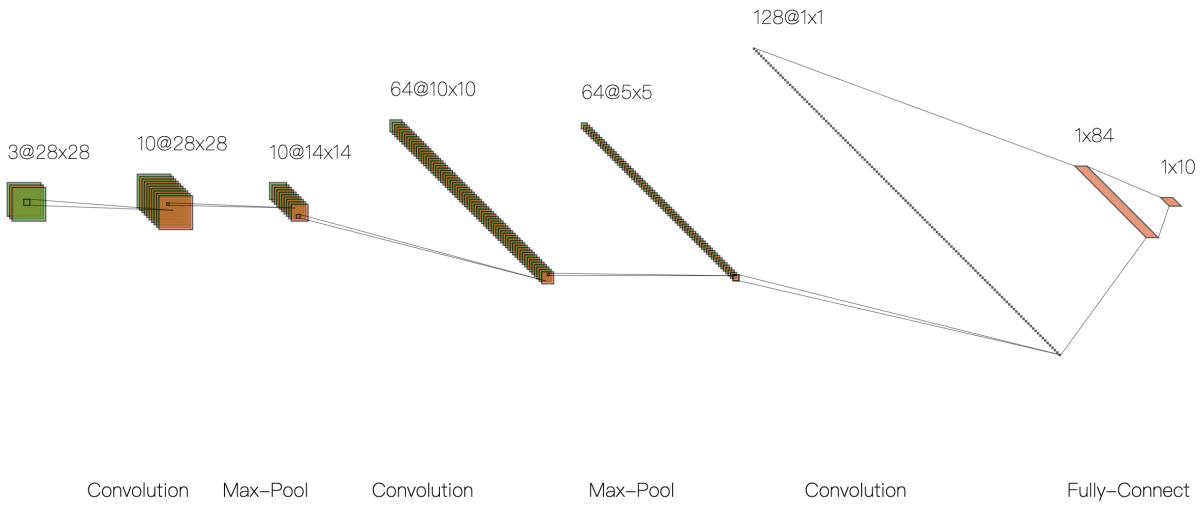

Figure 2: CNN Architecture

2.3 Workload Comparison

Now we conclude some key workload difference between CNN and LSTM. The largest difference is the degree of parallelism. As we pointed out before, CNN has more potential of parallel execution, while LSTM can only be processed sequentially given a sequence of input. Therefore, when we change model hyperparameters such as batch size, input and hidden dimensions, the training time of LSTM could be largely influenced. However, CNN will not be easily affected by them.

Another workload difference is their sensitivity to different input sizes. LSTM's execution time is largely dependent on input size (sequence length), because it needs to process the input sequence one by one. On the other hand, CNN's execution time is not easily influenced by varying input data size.

Furthermore, CNN is usually used in image-related tasks, while we often use LSTM to process text data. However, this is not always true and is not a decisive difference between CNN and LSTM. CNN can also be used in natural language processing applications, such as language modeling [8] and named entity recognition [2]. Meanwhile, LSTM can also be used in image classification, as we will describe in this paper.

3 Design

We have designed a CNN and a LSTM in order to characterize the workload and compare different performance between CNNs and LSTMs. In this section, we first describe the architectures of our CNN and LSTM separately. Then we introduce the technique we used to make these two models comparable when trained on the same dataset. Finally, we describe the key factors we considered when designing experiments and experimental setups.

3.1 Task and Dataset

We choose image classification as our benchmark task because both CNNs and LSTMs have achieved good performance on it. We use MNIST as our benchmark dataset. MNIST is a hand-written digit database commonly used for training image processing systems. The images in this dataset are hand-written digits from 0 to 9. Given an input image, our model needs to predict which digit it is. Therefore, this is a 10-way classification task.

3.2 CNN Architecture

We designed a CNN model very similar to LeNet [13], as shown in Figure 2, with 3 convolutional layers, 2 max-pooling layers and 2 fully-connected layers. The input is an image of size 28×28, and the output is a k-dimensional vector where k is the number of total classes. The first convolution layer uses the "same" padding strategy (maintaining the original size), while the other two convolution layers use the "valid" padding strategy (decreasing image size). The kernel sizes are set to 5.

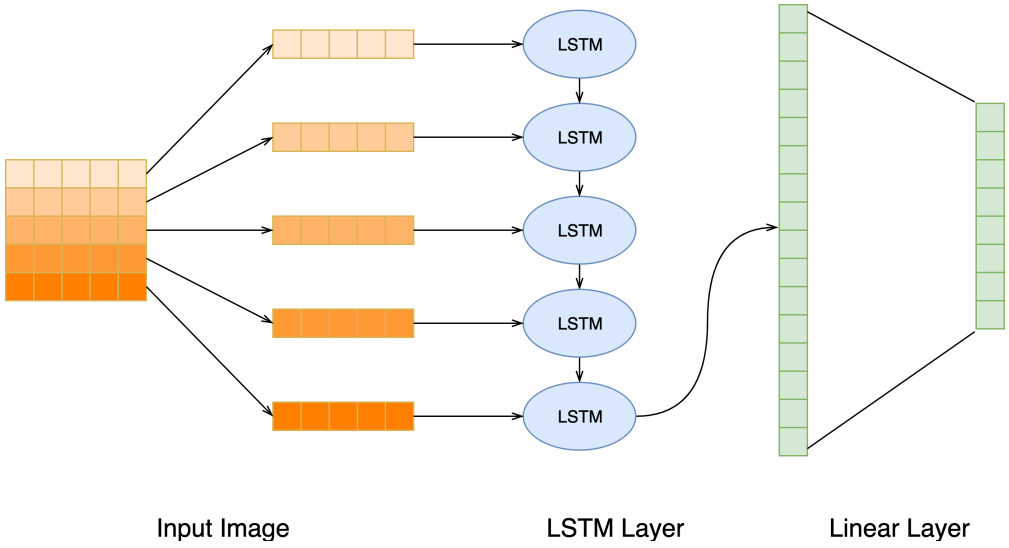

Figure 3: LSTM Architecture

3.3 LSTM Architecture

As shown in Figure 3, we implemented a one-layer LSTM model. We view a row of pixels as one elment, and an input image is thus a sequence of row elements. The LSTM cell takes a row of pixels as input at each iteration. Therefore, the input dimension of LSTM is 28, and we set LSTM's hidden dimension to be 128. Then, we extract the last output element (a 128-dimensional vector) from the LSTM layer and apply a linear transformation on it in order to generate a k-dimensional vector where k is the number of total classes.

3.4 Balance CNN and LSTM Models

Since the architectures of CNNs and LSTMs vary a lot from each other, it is not trivial to make these two models comparable (w.r.t. size) and even judge whether these two models are comparable. There's no previous work in the literature exploring this question. In this work, we propose a heuristic approach to balance the size of our CNN and LSTM models by equalizing their floating-point operations (FLOPs). We manually tune CNN's feature dimensions and LSTM's hidden dimension and calculate the FLOPs of CNNs and LSTMs respectively, until we reach a stage where CNN and LSTM have roughly the same FLOPs.

After carefully tuning model hyperparameters, we finally obtain a configuration where CNN and LSTM are of comparable size: 4.52M FLOPs for CNN and 4.49M FLOPs for LSTM. The final configurations are described in the above two subsections.

3.5 Experiment Considerations

After designing the model architectures, we need to consider and carefully select what factors may cause CNN and LSTM to behave differently and can best help us characterize the difference between them. We come up with the following aspects based on which to carry out experiments:

Thread number Since CNNs and LSTMs have different degrees of parallelism, we could train our models with controling the number of threads. When we increase the number of threads, the performance of CNN should benefit more than that of LSTM.

Batch size Since CNNs have more potential to be executed in parallel, when we increase batch size, CNN should benefit more than LSTM. We will measure both training time and GPU utilization to see this.

Hidden dimension Same as before, CNN should suffer less than LSTM when trained with a large hidden dimension. We will measure both training time and GPU utilization to see this.

Input size Since LSTM cannot be executed in parallel on the input sequence, it should suffer more than CNN when the input size increases. We concatenate images vertically in order to increase the image size and sequence length. We will measure both training time and GPU utilization to see this.

Memory usage Since LSTM needs to store all the intermediate computation results generated in the LSTM cell at each input element, for later usage and backpropagation computation, we assume its memory workload will be different from CNN's. We will measure both memory bandwidth and cache misses to show this.

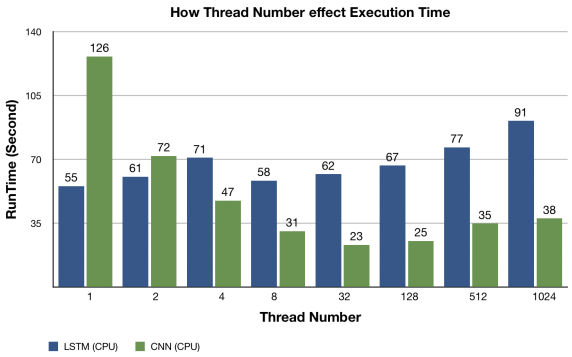

Figure 4: Thread Number Experiment

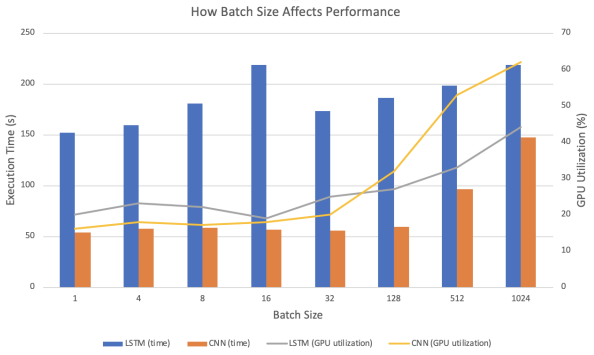

Figure 5: Batch Size Experiment

4 Evaluation

The experiment is run under the CloudLab machines. The hardware that carries out our experiment results are NVIDIA P100 GPU and Intel Xeon Silver 4114 Processor CPU with 10 cores, 768 Gb memory. We will now consider a series of examples that highlight the plurality of useful output patterns supported by the controlled settings of both CNN and LSTM models.

4.1 Thread Control Variation

Note the variation between LSTM and CNN models in Figure 4. As we increase the number of threads running in the system exponentially from 1 to 1024, the total execution time taken for CNN is largely shorten, while LSTM model does not respond much from the changes. Having the same configurations and machines, similar and however more drastic trend is also observed by tuning the size of the models to be much larger ones. Thus, we can conclude that LSTM cannot be paralleled as much as CNN in the same CPU configurations with same amount of workload.

To investigate the reason behind this disparity, we also examined the reason behind this is due to the back-end synchronization Tensorflow enforced on LSTM models. Since on each recurrent layer, RNN models like LSTM have to wait for the result from previous layer. We can also see this implementation in Tensorflow 2.0. Each forward pass of the LSTM cell will require the result from previous state.

In addition, we can see that the increased performance from the number of threads starts to turn around at about 32 threads. We assume that the reason for this reverse effect is that the maximum optimal threads for the CPU that we used is around 40. Thus, the increased communications from the increase of threads can counteract the benefit of increased parallelism.

4.2 Batch Size Variation

Note from Figure 5 that we include two dimensions that aiming to compare the performance of two different models. Having the batch size increases exponentially, we investigated the resulting variation of the total execution time and GPU cores utilization percentages.

In the same settings, LSTM's training time is almost 3X longer than the time taken for CNN model. However, we can observe that at batch size 32, the total execution time for LSTM model actually decreases. The reason behind this reverse effect may due to the lazy batch optimization that Tensorflow has since more data is being fed in for each iteration. For CNN model, we can see that the increase of batch size does not effect the execution time, which can be regarded as constant.

When increasing the size of each batch to 1024, we can also observe that the CNN model has reached its nearly 60% utilization. However, LSTM model is only using 40% of the maximum cores utilization. Thus, we conclude that there is still room for optimization of the LSTM model since it can be tuned to use more cores in parallel by using up the remaining cores.

4.3 Hidden Dimension Variation

During training, the size of the hidden layer of the neural network is also a another import hyper-parameter to consider when it comes to hardware utilization. The variation of the hidden layer in our experiment presents a nontrivial output.

Note from Figure 6, we can easily observe that LSTM generally would take more time to execute than CNN, scaling to almost 2X more. The execution time remains mostly stable and constant for all of our tuning. However, compared to CNN, LSTM starts to drastically increase after 512 numbers of layers. At 1024, we can see that the LSTM model takes almost 5X longer to execute than CNN. we also profiled the statistics of the cores utilization of GPU during training. We can see that the uti-

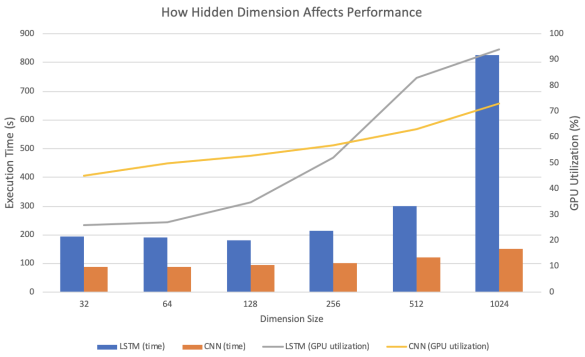

Figure 6: Hidden Dimension Experiment

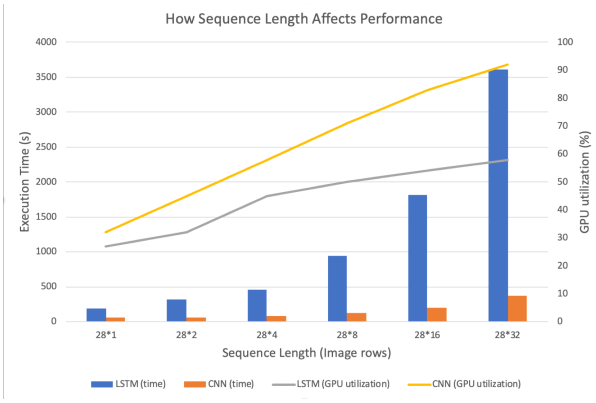

Figure 7: Sequence Length Experiment

lization rate is spiking very fast for LSTM, but not so much for CNN. At the tested maximum layer 1024, the LSTM already occupies 90% of the kernel with respect to the 75% of CNN. Thus, we can see that the LSTM is less optimized on the hardware platform or has reduced performance on GPU.

To investigate the reason behind this irregular behavior of LSTM, we also measured the memory utilization of both models. From the profiled statistics, we observed the almost 3X memory usage of the LSTM compared to CNN. Due to the characteristics of LSTM structure, several state metrics are retained throughout the training process. We suspect that the intensive usage of memory on hardware impedes the better performance of LSTM models.

4.4 Input Size Variation

In order to increase the workload of both of the models and further comparison, we examined the effect of varying the dimension of the training data set. Given that we feed both LSTM and CNN model with the popular Mnist training dataset, we decide to expand the matrices by making copies. As exemplified in the graph, 28 * N

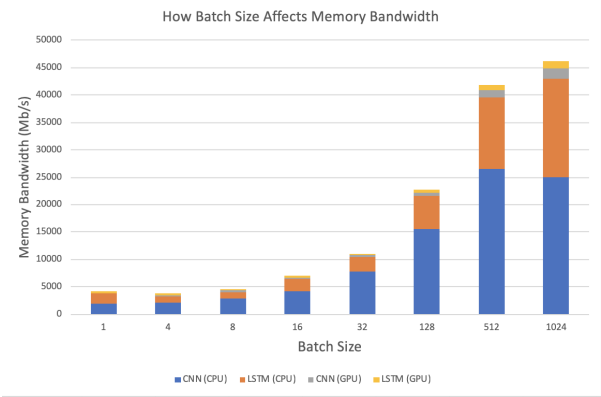

Figure 8: Memory bandwidth Experiment

(N = 1-32) indicates N copies. In this way, we are able to profile the performance in various layers of the neural network.

From Figure 7, we can see that LSTM's running time grows linearly with the exponential increase of sequence length. However, CNN's running time grows sub-linearly or constant with the increase of image size.

To investigate the reason behind this variation, we can make some comparisons accordingly. Note from 28 * 32 input dimension size, we can see that LSTM model is performing 10X slower than CNN model. In batch size variation and hidden dimension variation, we only saw a maximum of 5X difference. In addition, we can also see the GPU cores utilization of CNN is 2X higher than LSTM model. Thus, we can conclude that the increase of input size has the most significant effect (or worst) performance in LSTM model compared to CNN model under the same FLOPs computation.

4.5 Memory Variation

According to the measurements from the previous sections, we further investigate the reason for the limited computation power of LSTM by looking into the memory usage. We choose batch size as hyper-parameter to tune the experiments and got unexpected results.

From Figure 8, we can observe that the LSTM model is carrying 2X - 3X less memory bandwidth compared to CNN model. The result is non-trivial since LSTM model in general will take up more memory capacity or memory interaction due the various internal state variables. From the experiment result, we can see that LSTM model is not utilizing the memory as efficient as CNN model. We suspect that the limited optimization of memory usage of LSTM models in Tensorflow may be the dominant reason for many aspects.

5 Related Work

Existing benchmark primarily focus on solely running various deep learning models separately in varied environment with different machine and model parameter configuration. However, we compare the performance between LSTM and CNN under same FLOPS with different parameter settings. We used most of the same parameter variation such as batch size and compare CPU, GPU performance similarly like other related work.

While examining the effectiveness of resources usage of GPUs compared to CPUs, the work from Ousterhout et al.[16] has a significant implication for us in terms of performance measurement. We evaluate hardware usage statistics similiarly in a *blocked time analysis* fashion. However, we focus on GPU, memory and network performance instead of diving into the performance bottleneck of disk, network or CPU. The approach turns out to be relatively reasonable for identifying the hardware bottleneck since high parallelism leads to the heterogeneous resource usage, which may impair the accuracy of our measurement. In addition, Holmes et al.[6] also provides us insightful analysis into the recent development of GPU usage in neural network such as RNN or LSTM, suggesting the inefficient resource utilization of GPU either in academia research or industry production. Notably, from using the open source framework Tensorflow from Google, Tensorflow's GPU implementation has up to 90X higher latency than the CPU implementation on various model size on RNN. The surprising result leads us to investigate whether GPU may be the bottleneck for implementing recurrent neural networks such as RNN or LSTM.

As the recurrent computation process implied, RNN (LSTM) models require sufficient physical memory units to be optimally trained. However, we have only seen a slow improvement of hardware performance, [10] the latest PCIe improved only by a factor of 1X (PCIe gen3), contradicted to the larger and more complicated machine learning model in recent years. Thus, it leads us to the further research of the possible memory bottleneck for GPUs compared to CPUs. As aforementioned, Tensorflow [21] introduces us the remarkable memory mapping mechanism to fully utilize machine memories. Since the need for effective utilization of memory, it attracts us to further investigate whether memory becomes a bottleneck during training.

In distributed setting, communications between inter-rack relies on high performance of networking bandwidth. Work from Sapio et al.[18] has indicated similar concerns on the network bottleneck for distributed machine learning. In their work, the time for distributed training speeds up over 300% with modifications on the network. Thus, it leads us to further examine the closest network performance in Tensorflow via data parallel and model parallel. Compared to Sapio et al.'s work, we focus on profiling the network usage on various controlled variables.

In terms of benchmark evaluation, **ParaDnn** [19] is a first parameterized benchmark for end-to-end deep learning models. **ParaDnn** incorporates various hardware utilization analysis in heterogeneous deep learning environments, which provides us abundant benchmarking implications. In contrast, our work narrows and specializes to specific area which offers more in-depth analysis for further system designs.

DAWNBENCH [3] is another one of the state-of-the-art performance benchmark, which specialize in machine learning tasks. It was the first multi-entrant benchmark to use the TTA (*Time to Accuracy*) metric. The TTA metric also offers us insights on how to fairly evaluate the hardware bottleneck in deep learning jobs. In our work, we critically integrate such idea and further to ensure the unbiased, fair and accurate benchmarks of our research.

6 Conclusion and Future Work

We investigated the effect of varying number of threads, sequence length, memory bandwidth separately on LSTM and CNN model in broad range and profiled the performance of GPU and CPU separately.

Most of the experiment results that we currently have are based on execution time. We intend to find more correlation of other aspects such as hardware utilization rates in depth such as cache hit/miss rate. In our report, we have shown that LSTM has more limiting computation power (optimization) towards conventional hardware accelerators compared to CNN training. Thus, we suspect that LSTM models will have less desiring power.

For future work, we are interested in the profiling under distributed setting, which may have much different result. In addition, we only deploy a single framework, TensorFlow, in our experiment. Other frameworks may have different results

References

[1] M. Artetxe, G. Labaka, E. Agirre, and K. Cho. Unsupervised neural machine translation. *arXiv preprint arXiv:1710.11041*, 2017.

[2] J. P. Chiu and E. Nichols. Named entity recognition with bidirectional lstm-cnns. *Transactions of the Association for Computational Linguistics*, 4:357–370, 2016.

[3] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré,

and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *CoRR*, abs/1806.01427, 2018.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[5] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[6] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference*. ACM, 2019.

[7] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[8] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush. Character-aware neural language models. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[10] Y. Kwon and M. Rhu. Beyond the memory wall: A case for memory-centric HPC system for deep learning. *CoRR*, abs/1902.06468, 2019.

[11] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[12] G. Lample, M. Ott, A. Conneau, L. Denoyer, and M. Ranzato. Phrase-based & neural unsupervised machine translation. *arXiv preprint arXiv:1804.07755*, 2018.

[13] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[14] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.

[15] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, June 2015.

[16] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, May 2015.

[17] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, Jun 2017.

[18] A. Sapio, M. Canini, C. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *CoRR*, abs/1903.06701, 2019.

[19] Y. Wang, G. Wei, and D. Brooks. Benchmarking tpu, gpu, and CPU platforms for deep learning. *CoRR*, abs/1907.10701, 2019.

[20] S. Xu, H. Zhang, G. Neubig, W. Dai, J. K. Kim, Z. Deng, Q. Ho, G. Yang, and E. P. Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 937–950, Boston, MA, July 2018. USENIX Association.

[21] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, M. Isard, M. Kudlur, R. Monga, D. Murray, and X. Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 18:1–18:15, New York, NY, USA, 2018. ACM.