Performance Analysis of Just-in-Time Compilation for Training TensorFlow Multi-Layer Perceptrons

Richard Neill, Andi Drebes, Antoniu Pop
School of Computer Science
The University of Manchester, Manchester, United Kingdom
Emails: *first.last*@manchester.ac.uk

I. Introduction

The TensorFlow system [1] has been developed to provide a general, efficient and scalable framework for writing Machine Learning (ML) applications. With the rapid advancement and popularity of ML, and deep artificial neural networks in particular, TensorFlow has found success in both research and industry, particularly as the main ML engine for Google's Cloud Platform [2]. TensorFlow targets a large, diverse range of application areas, from ML research on a single machine, to large, distributed, and heterogeneous production environments. It is therefore crucial that TensorFlow is able to achieve high performance in production executions, while also retaining its capability for fast and efficient prototyping. Furthermore, TensorFlow's userbase encompasses a wide variety of expertise in application performance optimization, thus the TensorFlow run-time must achieve these goals without overly burdening the user with complex performance concerns.

However, TensorFlow presents a very large parameter optimization space to the user, where programming and configuration decisions can have a significant impact on performance. Similarly to other popular ML run-times with high-level language interfaces [3], [4], [5], TensorFlow places a high degree of emphasis on programmability and portability, which then forms a trade-off with performance. To achieve high performance, programmers need to make decisions ranging from the choice of a parallelization strategy for ML models, to allocating computation to hardware resources, to the format, representation, and communication patterns of input and output data, as well as a host of options around operation compilation and fusion. These decisions are non-trivial, as they are highly dependent on the nature of the interaction between the ML model and the hardware and software configuration of the underlying system. To avoid this complexity, programmers may leave some decisions to the TensorFlow run-time, e.g., by not explicitly specifying the mapping of computations to the hardware. However, there is no automatic way for the run-time to reliably and consistently apply optimal mappings. This often results in the user engaging in an inefficient parameter search when attempting to optimize a particular TensorFlow application, further complicated when considering the potential to trade ML model accuracy for increased run-time performance.

It is our position that performance analysis tooling for TensorFlow could greatly aid this process, and may potentially lead to automated program optimization techniques. As a first step, in this extended abstract, we investigate the performance of Just-In-Time (JIT) compilation in TensorFlow for the relatively straightforward use-case of training Multi-Layer Perceptrons (MLPs). By employing performance analysis, we aim to develop an understanding of when JIT compilation may be beneficial for performance, which could then be used to enable or disable JIT compilation in future program executions. We then suggest additional performance analysis techniques that could be employed in order to extend and automate the process for optimizing TensorFlow, in particular with respect to JIT compilation.

In the remainder of this extended abstract, we first overview JIT compilation in TensorFlow in Section II, before describing the evaluation process and some of our initial results in Section III. We conclude and outline future work in Section IV.

II. JIT Compilation in TensorFlow

Just-in-Time (JIT) compilation has been implemented in TensorFlow as a method to achieve fast program execution without sacrificing programmability and flexibility. JIT compilation occurs at execution time to dynamically compile interpreted TensorFlow code into highly-efficient, optimized machine code. By dynamically compiling at execution time, the TensorFlow run-time has enough information about the target code without requiring the programmer to specify its exact nature up front. However, the process of compiling code at execution time itself takes time, meaning that there is a trade-off between the performance overhead involved in JIT compilation and the performance benefit afforded by the optimized code. In this section, we briefly introduce the computational model of TensorFlow necessary to outline the JIT compilation process, and refer to [6] for a detailed description.

TensorFlow represents the computation specified in the user program as a *dataflow graph*, where nodes represent *operations* and directed edges between nodes represent the ordered flow of data that the operations apply to. Data is represented by *tensors*, where a tensor can be considered as a generalized matrix of a given dimensionality. Each operation describes a particular transformation of its input tensor(s) to produce an output tensor. Example operations include standard

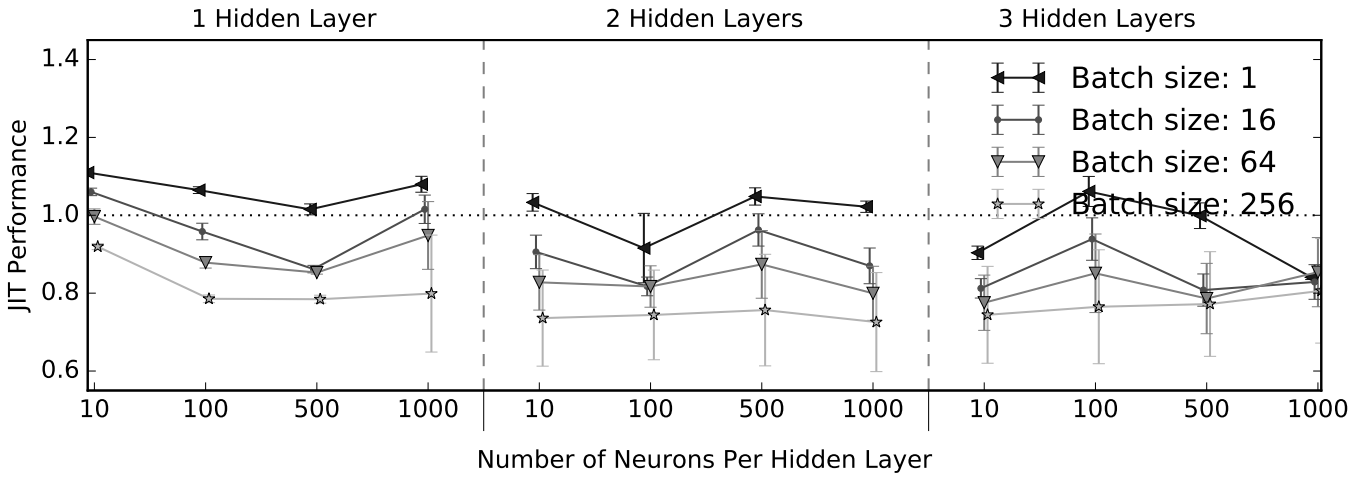

Fig. 1. Performance of training MLPs with JIT compilation enabled, relative to training equivalent MLPs where it was not enabled. Values greater than 1.0 therefore represent a performance increase given by JIT compilation. Data points are slightly shifted in the horizontal axis to visibly differentiate error bars.

mathematical computations such as matrix multiplications and summations, as well as heavily-used, specialized computations for ML applications, such as RelU activation [7] or batch normalization [8].

TensorFlow's graph-based computational model allows for a high degree of flexibility in the run-time's optimization of the program, through the application of graph transformations. First, the computation graph is partitioned into a set of subgraphs that are allocated for execution on available hardware resources, which may be distributed and/or heterogeneous. Dataflow edges between the subgraphs are then transformed into a set of data communication operations between the relevant devices and each subgraph pruned and optimized for computation, e.g., using JIT compilation, described next. Once the subgraph has been processed, the code is finally executed by the worker threads associated with each device.

To carry out JIT compilation, for each subgraph, the valid operations that the user has selected for JIT compilation, if any, are analyzed and grouped into *clusters* of connected nodes. For each cluster, the TensorFlow operations are converted to an intermediate representation (IR) language called 'High Level Optimizer' (HLO). This decomposes the operations into a set of more primitive HLO operations, for compilation via Google's Accelerated Linear Algebra (XLA) compiler [9]. The XLA compiler may then run optimizations on the HLO operations, before emitting LLVM IR to be compiled by a *backend compiler* for loading onto the target device, e.g., NVPTX to compile CUDA kernels for NVIDIA GPU execution, or an LLVM backend targeting the appropriate CPU architecture. Finally, once each cluster has been compiled for the target device, the nodes of each cluster in the TensorFlow subgraph are replaced by individual calls to the respective compiled binaries.

Using JIT compilation for TensorFlow can improve the performance of each cluster in the subgraph in two ways. First, optimization passes of both the HLO IR and LLVM IR can generate more efficient native code for each original HLO primitive operation. Secondly, multiple adjacent HLO primitive operations may be *fused* into single, efficient computations. However, the overhead of clustering TensorFlow operations, transforming the clusters to HLO, then generating LLVM IR, optimizing, compiling, and updating the TensorFlow subgraph may be significant. In the next section, we describe our initial investigation into whether or not JIT compilation in TensorFlow improves or impairs performance when training MLPs.

III. Performance Analysis

We analyzed the effect of enabling TensorFlow's automatic JIT compilation on the performance of training MLPs. An MLP is a common, but relatively straightforward feed-forward artificial neural network design that is composed of a series of fully-connected layers. For the experiment, we constructed multiple MLPs with an input layer consisting of 13 neurons and an output layer consisting of a single neuron. To investigate the effect of larger networks, and thus greater computational requirements for training, each MLP was configured with different hidden layer architecture, with the number of hidden layers ranging from 1 to 3, and widths ranging from 10 neurons to 1000 neurons. To simplify this process, the architectures were constrained such that all hidden layers consisted of an equal number of neurons. Further, the MLPs were trained across different batch sizes ranging from 1 to 256, with each training execution running the Adam algorithm [10] for 20 epochs, optimizing the loss function as mean-squared-error. Each MLP architecture and batch size configuration was independently constructed and trained twice, first without any JIT compilation, then with JIT compilation enabled via the flag `global_jit_level` set to `tf.OptimizerOptions.ON_1`. The training times were then compared to quantify the relative performance impact of JIT compilation.

Our experimental system consisted of an AMD A10-7890K CPU with 2 physical cores and 4 hardware threads, 16 GB main memory, running a NVIDIA Titan V GPU. We used the latest TensorFlow version `v1.5.0-2285-g4448430`, where the `global_jit_level` currently invokes the JIT process only for subgraphs allocated to GPU devices. However, enabling automatic global JIT compilation in this way is likely the first and most straightforward step that a programmer may make to optimize their TensorFlow program with compilation.

The results are shown in Figure 1, showing the relative performance of training with JIT enabled compared to training without JIT for the different configurations. These initial results indicate that very few MLP architectures benefit from JIT compilation, where its overhead can be seen to cause an increase in the overall execution time for most of the training runs. In general, the results suggest that JIT compilation has a more detrimental effect as the depth of the MLP increases. Whilst JIT compilation in TensorFlow is currently work-in-progress, these results are perhaps surprising. The overall computation requirement of training an MLP increases as the depth and width of the MLP increases, by incorporating additional but equivalent computations. Therefore, one would expect the performance benefit of JIT compilation to increase as MLP architecture gets larger. The results may therefore suggest an issue with TensorFlow's ability to cache compiled computations, additional overhead involved in the calls to the external compiled binaries, or inefficiencies in the generated operation kernels themselves. The results also show that JIT compilation harms training performance more for larger batch sizes. Specifying larger batch sizes results in much more parallelized computation by incorporating more elements into each matrix-multiply operation. Standard TensorFlow uses highly optimized linear algebra libraries such as NVIDIA's cuDNN for GPU, and Eigen or Intel's MKL-DNN for CPU. This means that the ability for JIT compilation via XLA to outperform these libraries is likely diminished as the batch size increases.

The results clearly indicate that a more detailed analysis is needed to understand when JIT compilation should be enabled. Indeed, it is likely that to achieve optimal performance, JIT compilation must be configured at a much greater granularity - in order to compile just those operations that are most likely to be worth the overheads involved. In the next section, we conclude and outline some of the future research directions that we intend to investigate around the performance analysis of TensorFlow and JIT compilation in particular.

IV. Conclusion and Outlook

In this extended abstract, we discussed JIT compilation in TensorFlow and experimentally indicated that enabling it can be detrimental to program performance, depending on the nature of the application.

Moving forward, we intend to carry out more sophisticated analyses for enabling the optimization of TensorFlow programs. To do this, we propose to extend the instrumentation capabilities of TensorFlow, as the current framework for performance analysis focuses only on recording the finally executed operations. By further instrumenting the TensorFlow run-time itself, we aim to better understand the overheads involved in graph transformations and optimizations such as JIT compilation. Instrumentation at a higher granularity may then potentially enable the development of models to help users specify their program optimally, or indeed for the run-time to automatically decide which optimizations to enable, specific to the user application and system. Considering JIT compilation for example, it may be very useful to calculate the performance benefit at a per-operation level, as opposed to across the whole program execution as investigated in Section III. Moreover, as many ML applications apply similar computations to data, in future analyses we intend to more generally investigate the performance benefit of JIT compilation of common computational patterns such as TensorFlow and HLO operation clusters, to predict the performance benefit resulting from processing and fusing the operation sets. These techniques would allow for more dynamic, potentially automatic configuration of JIT compilation, and therefore higher performance of TensorFlow programs without additional demands on the programmer.

Acknowledgements: This work was supported by the grants EU FET-HPC ExaNoDe H2020-671578, UK EPSRC EP/M004880/1 and UK EPSRC EP/I028099/1. A. Pop is funded by a RAEng University Research Fellowship.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[2] G. LLC. Google cloud platform: Cloud machine learning engine. [Online]. Available: https://cloud.google.com/ml-engine/

[3] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[4] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[5] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.

[6] P. Goldsborough, "A tour of tensorflow," *arXiv preprint arXiv:1610.01178*, 2016.

[7] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[8] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[9] G. LLC. Xla - tensorflow, compiled. [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

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