Benchmarking PyTorch's and TensorFlow's Speed and Power Performance Building Classifiers on the MNIST Dataset

Joe Germino, Tim Inzitari, Nhat Le
University of Notre Dame
Department of Computer Science & Engineering
{jgermino, tinzitar, nle5}@nd.edu

***Abstract*— Deep learning is a mainstay of modern-day classification tasks. There are a wide variety of deep learning packages publicly available including some developed by leading tech companies such as Google and Facebook. These packages ease the process of developing complicated networks for different tasks. In order to understand which tools are best for a given task, it is imperative to understand the strengths and weaknesses of each from a systems performance standpoint. In this paper, we will attempt to benchmark the performance of two major frameworks within Python: PyTorch and TensorFlow. We will compare their performance across a variety of machines in terms of both their speed and power efficiency by training and testing a K-nearest neighbors classifier, a simple Neural Network, and a deep Convolutional Neural Network using the MNIST dataset. We will measure speed efficiency by measuring time to train over a series of tests.**

I. INTRODUCTION

Modern progress in deep learning has given a rise to machine performance in a variety of tasks, including, but not limited to image classifications, image segmentation, and natural language translation [1]. To better develop and deploy deep learning models quickly and effectively, multiple frameworks have been created. Leading tech companies and research institutions have been developing deep learning networks for industrial and research purposes: Google with TensorFlow [2], Meta with PyTorch [3], and Berkeley with Caffe [4], for example.

One of the most common tasks that deep learning models try to achieve is classification. In the image domain, the MNIST dataset [5] is often used as a benchmark to evaluate a model's performance. The dataset consists of gray-scales images of handwritten digits, split into training and testing sets. The goal of a model on this dataset is to identify which digit appears in the image. This dataset does not require pre-processing efforts, therefore, it is a good starting point for researchers to evaluate their classification models.

Developing wide and deep networks to achieve state-of-the-art performance in a variety of tasks has been the main focus of the deep learning community for a while. However, to make deep learning ideas not simply an academic curiosity, more attention should be paid toward the performance metrics of these models. With that in mind, in this paper, we aim to compare the two most dominant frameworks in the community, TensorFlow [6], and PyTorch [3]. Having developed different models in both frameworks, including a K-nearest neighbors classifier, a simple neural network with a few hidden layers, and a complex deep neural network utilizing convolutional layers, our contributions consist of comparisons between the training time and power usage of the models in both frameworks. In all cases, the work done in this paper utilizes the MNIST Handwriting dataset [5].

Another important development in recent years comes from hardware not software. Recent developments in CPU, GPU and System on a Chip (SoC) engineering tactics have lead to increased optimizations across the board in all of computing, not just specifically deep learning. [7] The opening of the CUDA architecture by NVIDIA as well as simple architecture increases by Intel and AMD in processors in recent years have led to massive increases in many aspects of matrix level computing. One of the most significant changes in recent times is the onset of laptop and desktop level SoC's, a system typically reserved for mobile chips, which is being flagshipped by Apple and their M1 chip for their transition to "Apple Silicon". [7].

The difference between a SoC and other architectures is that the SoC combines multiple hardware objects into single areas, such as GPU and CPU, as well as other potential optimizing areas. Figure 1 [8], is from Apple's diagram of their ARM based M1 chip during their press release for the 2020 release.

The M1 contains numerous aspects other than a simple GPU, CPU, and RAM set up. One item that is

Fig. 1: Apples press diagram of M1 Architecture, from 2020, November 10 Apple Event announcing its release.

interesting is the Neural Engine and Machine Learning Accelerators. Important is the M1's architecture on the physical CPU die. It contains 8 cores, 4 of which are what they call "high performance" and 4 are "efficiency". At the 2020 Apple Event [8] they claim that an "efficiency core" is around half as powerful as the high performance core. To aid users in taking full advantage of the M1 chip they released documentation for users to implement their "ML Compute" Framework into applications [9]. These extra applications paired with the fact that all ARM MacOS Systems will be running an M1 or similar chip, has led Apple to tightly couple the operating system and hardware system in order to maximize performance.

The layout of this paper includes previous related work in section 2, our approach in section 3, and the results of our contributions in section 4.

II. RELATED WORK

Many deep learning products and research have been created and conducted using different combinations of hardware and software. For the hardware side, the most dominant processors are GPU and CPU. For the software side, the most common frameworks are TensorFlow and PyTorch created by Google and Meta respectively. [10]

The main difference between a CPU and a GPU is the number of cores. While CPUs have fewer cores, each core is much faster and much more capable. As a result, it is great at sequential tasks. GPUs, on the other hand, have more cores; however, its cores are much slower, making it great for parallel tasks. [10]

PyTorch and TensorFlow utilize CPUs and GPUs to run the process. These deep learning frameworks ease the process of creating complicated deep learning models by abstracting different model layers and automating the process of calculating the gradient. The main difference between the two is that PyTorch adapts a dynamic graph approach, meaning that the graph structure is defined on-the-fly via the actual forward computation [11] whereas TensorFlow adapts a static graph approach, meaning that the users define the graph completely and then inject data to run [2].

Many deep learning architectures and Machine Learning algorithms have been evaluated on the MNIST dataset, both in TensorFlow and PyTorch [12]. In [13], Ketkar gives an overview of PyTorch as a tool to develop deep learning models. Similarly, in [6], Abadi et al. provide an overview of the TensorFlow framework. Using these frameworks, Siddique et al., developed different CNN with varying number of convolutional layers and compare their performance [14]. Most similar to our work, Jain et al. explored training Deep Neural Networks across multiple CPU architectures using both frameworks [15]. The authors evaluated the performance characterization of deep network models on different CPU architectures. In this work, we will focus on the performance of different networks built by both frameworks, including training time and power efficiency on different hardware systems: AMD Ryzen 5 3600x CPU, Nvidia RTX 2070 Super GPU, and Apple's M1 SoC 2020 MacBook Pro.

III. APPROACH

Our approach to testing the efficiency of PyTorch and TensorFlow involved building a K-Nearest Neighbors, a simple Neural Network, and a Convolutional Neural Network model on the MNIST dataset on each of three different hardware systems: an AMD Ryzen 5 3600x CPU, Nvidia RTX 2070 Super GPU, and Apple's M1 SoC 2020 MacBook Pro. Each of the models were built using code from prior work measuring accuracy on the MNIST dataset.[16] [17]

To evaluate the systems, we utilized two major evaluation methods: training time and power consumption. The accuracy of the models was a secondary priority as success was evaluated based on direct comparisons of these two metrics as we run the models on different hardware systems mentioned above.

For each of the three model types, we used a simple training and testing split of the dataset randomized for each iteration. This was chosen as a standard practice to help limit bias towards any architecture. Models were considered accurate when they correctly identified handwritten characters in the testing set that has not been included in the training set.

To measure the training run time, each model was built 20 times on each system. For each of these epochs,

run time was measured using built-in Python tools from the Time library. For each run, only the time to perform training is measured. Dataset filtering and other I/O's are not included in the measurement.

For each of these 20 models, predictions were made on the testing set to calculate a model's accuracy.

To measure power consumption we utilized the Running Average Power Limit (RAPL) system. [18] Originally developed by Intel this system has since been extended to support newer AMD Ryzen processors as well. RAPL was implemented to measure total Joules used by a given hardware component for the entire 20 epoch training cycle and then stored for later comparisons. For CPU runs, we measured the energy usage of the CPU, and for GPU runs we measured the power called by the GPU system. We were unable to measure M1 power usage.

IV. RESULTS

To compare the overall efficiency of the different systems, we evaluated the results using two different metrics: training time and power efficiency. A breakdown of each system's performance for each method is contained below.

Figure 2 displays the median training times for each system and framework along with the standard deviation and quartiles over the 20 Training epochs for the Simple Neural Networks. The corresponding mean and standard deviations are included in Table I while the quartiles underlying the graph are contained in Table II. Figure 3, Table III, and Table IV contain the same information for the Convolutional Neural Network training times.

Figure 4 and Figure 5 display the individual run times of each epoch for the different Frameworks on M1 and CPU, respectively. The key additions of these graphs are the inclusion of Scikit-learn's K-nearest neighbor algorithm as a baseline. More comment on this comparison is included in the analysis below.

Time (in seconds)	CPU	GPU	M1
PyTorch	51.29 ± 1.14	16.51 ± 0.17	89.55 ± 10.47
TensorFlow	8.62 ± 0.17	9.45 ± 0.30	17.83 ± 0.98

Table I: Average Training Times and Standard Deviations for Simple Neural Networks

A. Framework Comparison

TensorFlow significantly outperforms PyTorch across all systems. In all cases, the average run time in PyTorch is greater than 3 standard deviations (and

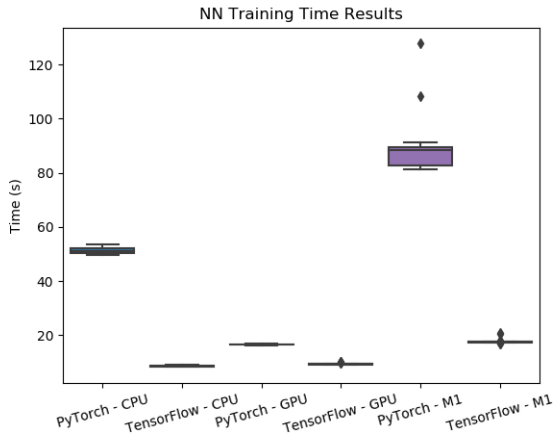

Fig. 2: Training times for 20 Simple Neural Networks on each Framework and Hardware system.

Time (in seconds)	Q1	Median	Q3
PyTorch - CPU	50.48	50.93	52.05
TensorFlow - CPU	8.49	8.61	8.74
PyTorch - GPU	16.45	16.51	16.62
TensorFlow - GPU	9.25	9.34	9.46
PyTorch - M1	82.80	88.39	89.45
TensorFlow - M1	17.41	17.53	17.66

Table II: Training Time quartiles for simple Neural Networks

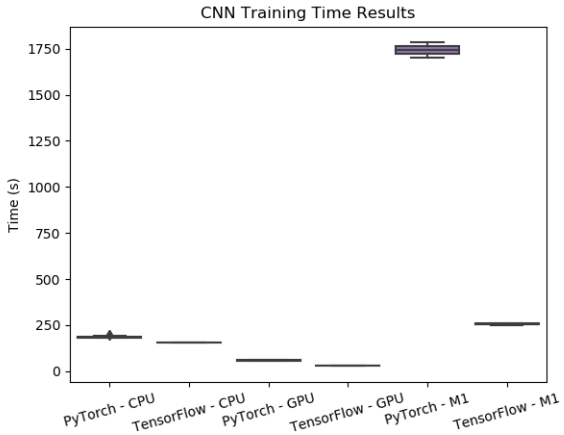

Fig. 3: Training times for 20 Convolutional Neural Networks on each Framework and Hardware system.

Time (in seconds)	CPU	GPU	M1
PyTorch	184.51 ± 4.67	58.48 ± 1.06	1744.37 ± 23.71
TensorFlow	155.13 ± 0.96	28.21 ± 0.29	256.13 ± 2.80

Table III: Average Training Times and Standard Deviations for Convolutional Neural Networks

Time (in seconds)	Q1	Median	Q3
PyTorch - CPU	181.23	182.25	185.84
TensorFlow - CPU	154.64	155.26	155.76
PyTorch - GPU	57.82	58.39	59.24
TensorFlow - GPU	28.02	28.26	28.45
PyTorch - M1	1723.97	1744.43	1764.14
TensorFlow - M1	253.74	256.05	258.25

Table IV: Training Time quartiles for Convolutional Neural Networks

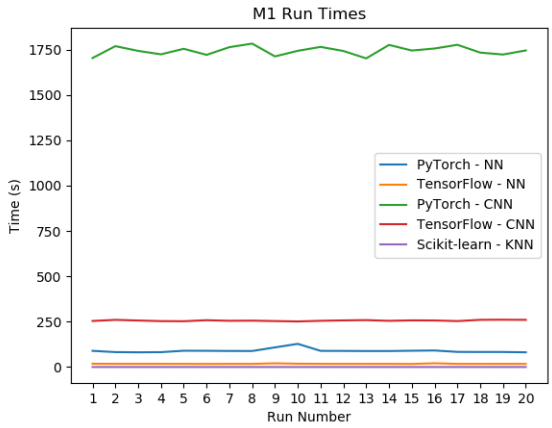

Fig. 4: Training times for each of 20 epochs for each Framework on M1.

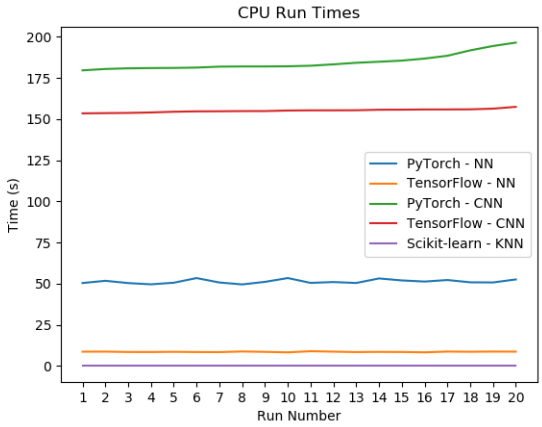

Fig. 5: Training times for each of 20 epochs for each Framework on CPU.

usually many more) away from the average run time in TensorFlow.

As the box plots indicate, PyTorch is frequently several orders of magnitude slower than TensorFlow. In every situation, the training time for TensorFlow falls well below the training time for PyTorch even when accounting for a margin of error.

Accuracy Per Model	Percentage
PyTorch NN	92.31 $\pm$ 0.00172
TensorFlow NN	95.83 $\pm$ 0.0025
PyTorch CNN	99.30 $\pm$ 0.0023
TensorFlow CNN	99.13 $\pm$ 0.0013
KNN	91.2365 $\pm$ 0.00

Table V: Accuracy Averages for Each Model Type

Further evidence of this is included in Figure 4 and Figure 5. These graphs demonstrate a clear and consistent order of training time for the different frameworks with TensorFlow always being faster than PyTorch. Notably, Scikit-learn's k-nearest neighbor implementation runs in virtually no time on both the CPU and M1 hardware. The framework is not compatible with GPU. This makes sense as KNN is a significantly simpler algorithm than NN and CNN. However, unlike the other frameworks, this increase in efficiency comes with a trade-off of a significant decrease in accuracy as displayed in Table V especially when compared against CNNs.

Power Usage	CPU Average	GPU Average
PyTorch NN	45993 $\pm$ 1013.7	21.350 $\pm$0.49252
PyTorch CNN	194720 $\pm$ 1823.3	68.989 $\pm$2.2430
TensorFlow NN	8500.3 $\pm$ 254.92	4.9401 $\pm$0.14866
TensorFlow CNN	182270 $\pm$ 6135.3	35.433 $\pm$0.90840

Table VI: Work Done Per Model

Additionally we can see in Table VI that in each situation the power usage for PyTorch implementations is higher than that of TensorFlow. This follows what should be observed based on theory, when you follow the Work equation of classical physics where Work (Joules) will be equivalent to the product of power and time. In each scenario above the operations are bound by processing power and all components should be running at near electrical capacity for the duration of the runs.

Finally, the bar plots in Figure 6 show the individual run times for each of the training epochs divided by hardware and framework. As seen with the CPU and M1 from earlier, these graphs clearly demonstrate that TensorFlow was consistently faster than PyTorch with zero instances of the TensorFlow training time being slower than a single similar PyTorch iteration.

When using the simple Neural Networks, TensorFlow sees an improvement on the CPU of nearly 6x

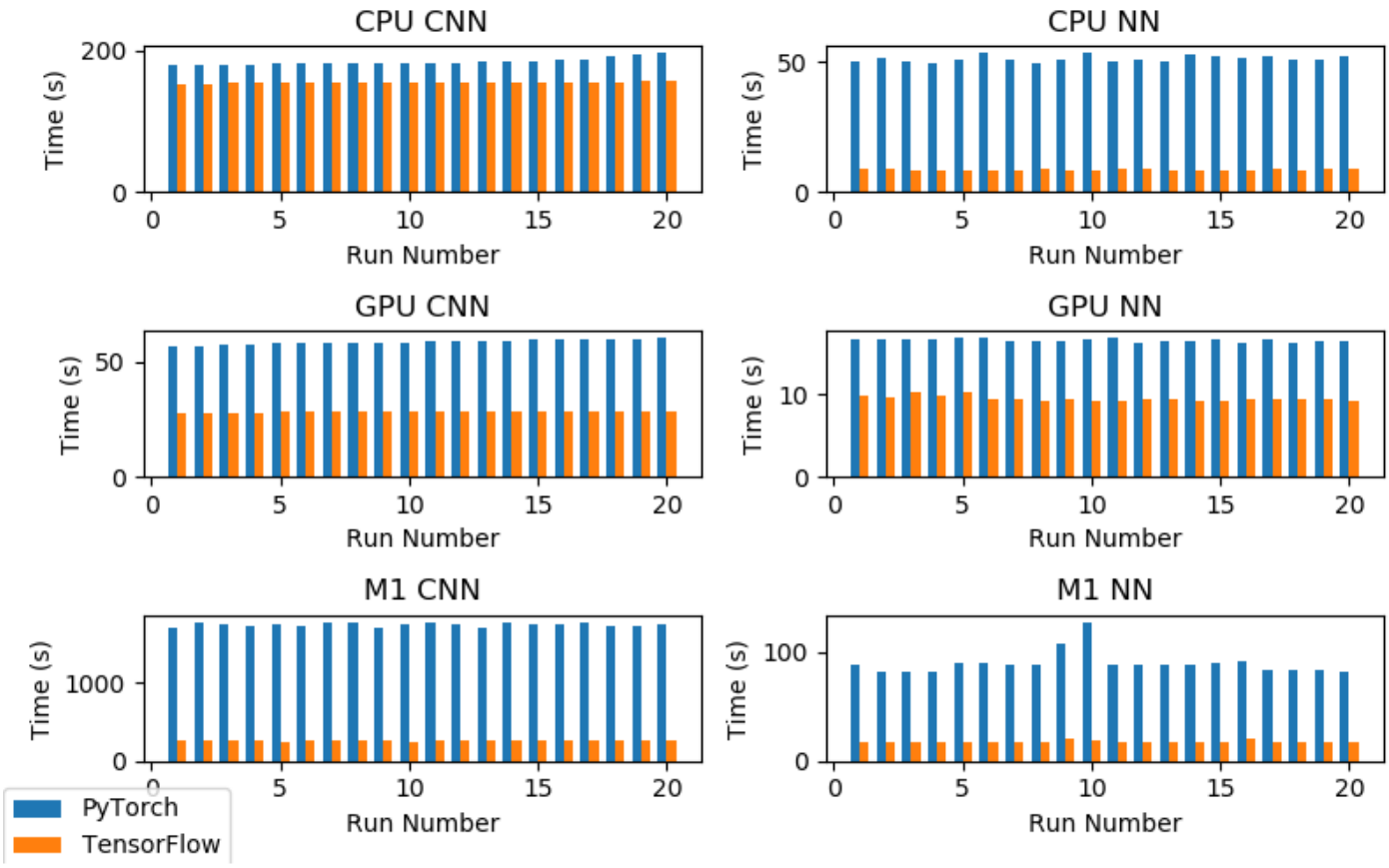

Fig. 6: Training Times for each epoch

compared to PyTorch compared to only 1.75x on the GPU. However, with the Convolutional Neural Networks, TensorFlow gives up some of its edge in CPUs while gaining ground in GPUs. Specifically, PyTorch is 1.2x slower on the CPU while over 2x slower on the GPU.

On the M1, TensorFlow is over 5x faster training a simple Neural Network while nearly 7x faster training a Convolutional Neural Network. As discussed above, given the tight coupling between TensorFlow and Apple, this result is expected.

In regards to accuracy, Figure 8 appears to show a clear correlation to be achieved with more complex models on this solution. The simple KNN model was fully deterministic revealing the same accuracy every time when using a k value of 5. Interestingly on the Neural Network, the base models given by PyTorch appear to be significantly less accurate than those given by default TensorFlow models. With the PyTorch system struggling to even beat out KNN models that run significantly faster. In the most complex CNN models however both PyTorch and TensorFlow are nearly indistinguishable at upwards of 99% accuracies.

The M1 results, however, are drastically different than other comparisons. This is with reason and will be discussed later in **M1 Differences**.

Notably, our results differed significantly from prior work specifically [15] which found evidence that PyTorch tends to outperform TensorFlow on GPUs. We believe the reason for this discrepancy lies in the version of TensorFlow which we were using. Our experiments were run using TensorFlow 2.0 which released in September 2019 after [15] was published. The updates to TensorFlow's framework appear to have significantly improved its performance on GPUs.

Overall, the results indicate that the more computationally expensive a problem is, the better TensorFlow is compared to PyTorch. In terms of training speed and power consumption, TensorFlow appears to have significantly less overhead and, in our experiments, is superior regardless of the parameters.

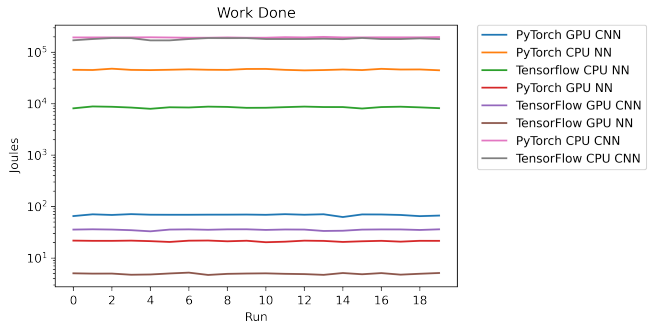

Fig. 7: Work Measurement Graph for Each Run

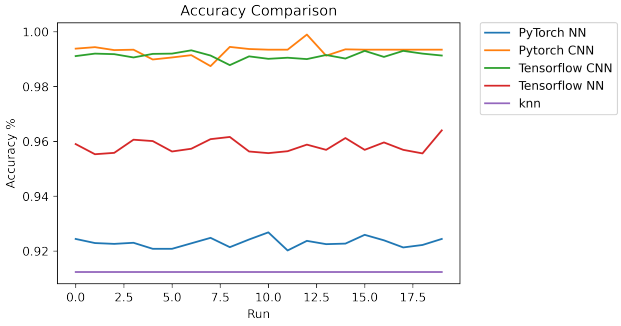

Fig. 8: Accuracy Graph for Each Run

B. Hardware Comparison

Given the nature of the MNIST classification problem and previous research such as [15], it is unsurprising that the GPU tends to significantly outperform both the CPU and M1. The lone exception is training the Simple Neural Network using TensorFlow in which case the CPU outperforms the GPU in terms of speed. This seems to be a consequence of TensorFlow's overall efficiency. While TensorFlow is still quick on the GPU, it is fast enough on the CPU that the additional overhead results in a decrease in performance. When training a much more complicated Convolutional Neural Network, the GPU is nearly 6x faster than the CPU suggesting that the improvement in performance outweighs any additional overhead.

For both the simple and Convolutional Neural Networks, the CPU outperforms M1. On the Simple Neural Network, PyTorch is approximately 1.75x slower on M1 than CPU while TensorFlow is approximately 2x slower. However, for the Convolutional Neural Network, TensorFlow is only 1.7x slower on M1 than CPU while PyTorch is 9.5x slower. TensorFlow's coupling with M1 appears to allow it to have the roughly the same relative training time to the CPU regardless of the problem. PyTorch, on the other hand, is not well equipped to handle large problems on the M1 resulting in extra overhead and a dramatic slowdown in training time.

In regards to power usage based on different hardware types, it appears to similarly be bound more strictly by the run time than the hardware being used. GPU's significant reduction in run time is also contributing to significant reduction in work being done by the system also following what classical physics would lead us to believe. Although no hard testing could be confirmed, it can also easily be inferred based on these results that the M1 power usages would be follow the same pattern and be mainly determined by the maximum running wattage and the run time of the SoC.

C. M1 Differences

The largest difference by far in the two systems comes on Apple's M1 SoC. PyTorch appears to take significantly more time than the TensorFlow implementation. For example, the difference is around 4x the time length on a simple Neural Network and around 8x on the CNN system.

Initially, this appeared to be an error due to PyTorch running in x86 architecture mode using Apple's Rosetta 2 system. This theory was debunked by deleting the Rosetta 2 system and achieving the same results leading to a deeper dive into the implementation of each architecture to arrive at the working theory.

The current theory is that TensorFlow's implementation is built with the M1 in mind and utilizes the full extent of the M1 SoC using the ML Compute framework, while PyTorch is a simple ARM compilation of its x86 counterpart. This theory is supported by TensorFlow being implemented for the Mac by Apple itself [19] and the time increases on PyTorch during outlier experimental runs on Neural Network experiments.

The significance by Apple developing the M1 port for TensorFlow is fairly obvious, they are in the best position to have the fullest knowledge of incorporating the M1 SoC's systems into code, since they created the chip. While a third party such as Meta and PyTorch would have to learn the documentation for ML Compute and implement it using that alone - a much more tedious task.

The other piece of evidence is less obvious, noticeably the M1 Neural Network runs have 2 outlier runs that took significantly more time than the rest of the runs, specifically runs 9 and 10 in Figure 6. According

to timestamps, run 9 for PyTorch began at 09:59:52am and would have still been running at 10:00:00am. The working theory is that when the time reached top of the hour some part of the system began doing update checks and these checks did not conclude until the TensorFlow run of Run 10. As explained at Apple's M1 Event [8] these updates are sent to the M1 CPU's efficiency cores with priority. Interestingly, run 10 of PyTorch which was completely in the time frame of these suspected updates has a time increase of almost the exact 50 percent that would be thought to occur theoretically by losing access to efficiency cores on a pure CPU system. While run 9 of TensorFlow, also within the update time frame meets a time increase of far below the expected 50 percent increase. This time discrepancy is evidence in support of our theory that the PyTorch system is a simple ARM port that runs on only the CPU of the M1, while the TensorFlow implementation takes full advantage of the M1 SoC. During the application updates, the PyTorch system lost access to its 4 efficiency cores and resulted in lower pluralisation performance increases nearly exact to what would be expected theoretically. While TensorFlow also lost access to these 4 cores it was not as heavily effected.

V. CONCLUSION/FUTURE WORK

In all scenarios, users should see significantly less time usage on a TensorFlow system when compared to a PyTorch system. This is best shown using an SoC system such at Apple's M1 Macbook.

Areas for further research would include testing on other hardware configurations. With the recent release of the M1 Pro and M1 Max SoC's it would be interesting to see how their performances compare to this system. Another area of further research would be further exploring the differences between ARM port theory for PyTorch on the M1 Mac, specifically gaining more test runs that cross events that take priority over the efficiency cores and a deep dive into the actual code implementation for the system to see if they attempt to use the ML Compute framework.

Another area of research that could be performed is their performance on different hardware architectures such as a Tensor Processing Units that have been emerging in the field in recent years. It can be explored how these chips highly optimized for machine learning calculations could improve processing power.

Additionally we could look into other machine learning applications besides classification, such as natural language processing or regression models to compare performance between architectures there.

REFERENCES

[1] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, pp. 27–48, 2016.

[2] 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 *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[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*, pp. 675–678, 2014.

[5] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[7] "Apple unleashes m1." `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`, Oct 2021.

[8] Apple, "Apple event — november 10." `https://www.youtube.com/watch?v=5AwdkGKmZ0I`, Nov 2020.

[9] "Ml compute — apple developer documentation." `https://developer.apple.com/documentation/mlcompute/`.

[10] F.-F. Li, "Deep learning software," 2017.

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[12] M. Abadi, M. Isard, and D. G. Murray, "A computational model for tensorflow: An introduction," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2017, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2017.

[13] N. Ketkar, *Introduction to PyTorch*, pp. 195–208. Berkeley, CA: Apress, 2017.

[14] F. Siddique, S. Sakib, and M. A. B. Siddique, "Recognition of handwritten digit using convolutional neural network in python with tensorflow and comparison of performance for various hidden layers," in *2019 5th International Conference on Advances in Electrical Engineering (ICAEE)*, pp. 541–546, 2019.

[15] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. Panda, "Performance characterization of dnn training using tensorflow and pytorch on modern clusters," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–11, 2019.

[16] "Training a classifier." https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

[17] "Training a neural network on mnist with keras tensorflow datasets." https://www.tensorflow.org/datasets/keras_example.

[18] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 189–194, 2010.

[19] Apple, "Tensorflow for macos accelerated using apple's ml compute framework." https://github.com/apple/tensorflow_macos.