Work-In-Progress: Making Machine Learning Real-Time Predictable

Hang Xu and Frank Mueller, North Carolina State University, USA, mueller@cs.ncsu.edu

ABSTRACT

Machine learning (ML) on edge computing devices is becoming popular in the industry as a means to make control systems more intelligent and autonomous. The new trend is to utilize embedded edge devices, as they boast higher computational power and larger memories than before, to perform ML tasks that had previously been limited to cloud-hosted deployments. In this work, we assess the real-time predictability and consider data privacy concerns by comparing traditional cloud services with edge-based ones for certain data analytics tasks. We identify the subset of ML problems appropriate for edge devices by investigating if they result in real-time predictable services for a set of widely used ML libraries. We specifically enhance the Caffe library to make it more suitable for real-time predictability. We then deploy ML models with high accuracy scores on an embedded system, exposing it to industry sensor data from the field, to demonstrates its efficacy and suitability for real-time processing.

KEYWORDS

Edge Computing, Real-time Predictability, Keras, Caffe

ACM Reference Format:

Hang Xu and Frank Mueller, North Carolina State University, USA, mueller@cs.ncsu.edu. 2018. Work-In-Progress: Making Machine Learning Real-Time Predictable. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND MOTIVATION

Rapid developments in embedded systems hardware and communication technology have made it feasible to deploy data analytics task on "edge" devices. These compute-intensive tasks were previously constrained to execute on cloud systems with much more powerful computing capabilities. As a result, control systems in the automotive and power grid areas can benefit from advances in machine learning on big data analytics. Traditionally, an industry data analytics scenario starts from the source data collected by sensor arrays deployed in the fields, then travels through intermediate networks and ultimately supplies input data to analytics algorithms running on a cloud computing system. The new trend is to deploy edge devices close to the data inputs so that edge computing tasks may directly process sensor input data with much lower latencies compared to first transmitting data over the network to the cloud. Benefits of such edge computing include:

• Streaming data inputs may be prohibitively large when directly transferred to the cloud. Nowadays, raw streaming data can be generated at a sampling rate over 500Hz by sensors [12], which severely impacts both communication bandwidth in transit to and computational requirements within the cloud. Considering the large number of data sources deployed in the field, directly transferring raw streams of data into the cloud would have prohibitive side effects on the stability of the Internet as such.

• Streaming data may have real-time performance and predictability restrictions in analytics. In industrial anomaly detection systems, the real-time performance of a task is expressed in terms of the ability to provide tight and predictable upper bounds on the task's worst-case execution time (WCET) [13]. These WCET bounds can subsequently be utilized to monitor task execution and detect timing anomalies due to malfunctioning hardware/software or cyber attacks [14, 15] — and to then respond to such anomalies by entering into a fail-safe state [5].

• Data privacy concerns restrict the transfer and use of data on the cloud. Customers as well as industries have been increasingly concerned about data privacy and may not trust that their data, when transferred to the cloud, remains private (for individuals) or can be guarded from competitors (for industries).

Among the analytics technologies of interest to industry is ML, which has been proved to be powerful to solve numerous problems [6]. While today's edge devices are comprised of a diverse set of computing devices, e.g., workstations, personal computers, microcomputers and micro-controllers, our research focuses on the latter, namely embedded systems hardware such as microcomputers and micro-controller. Yet, real-time performance and predictability of ML libraries on such embedded systems, particular on low-end power-conscious devices, has not been researched much to the best knowledge.

Contributions

This work makes the following contributions:

• We identify problems in ML processing on embedded systems.

• We assess the real-time predictability and suitability of widely used ML libraries with respect to embedded systems subject to timing constraints.

• We enhance the Caffe library to make it more suitable for real-time predictability.

• We deploy ML models with high accuracy scores on an embedded system, exposing it to industry sensor data from the field, to demonstrates its efficacy and suitability for real-time processing under the constrained resources of embedded systems.

2 MACHINE LEARNING ON EMBEDDED SYSTEM

Let us identify what kinds of ML problem are appropriate to be solved on an embedded system. We will initially consider supervised and unsupervised learning.

2.1 Training vs. Predicting

A typical ML task generally consists of two phases, training and inference. We argue that in a practical industry use scenario, timing

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

analysis for ML on an embedded system is focused on the inference phase for the following reasons:

• Computational resources on an embedded system are less powerful than on commodity computing systems, and ML *training* is more time demanding than the *inference* phase. Nowadays, a training task may require a high performance computing (HPC) system with hundreds, thousands, or more CPU cores to compute for several days in parallel using petabytes, exabytes or even more memory storage [8]. Such large resource demands for the training phase make it infeasible to assign training tasks to an embedded system with at most 8 cores and gigabytes of memory. And the gap between the demand by ML tasks and the resource supply of embedded systems is only increasing.

• Except for online ML tasks, it is not necessary to combine the training and inference phases on the same machine. For offline ML tasks, model development and its deployment can be separated. One may still periodically update the deployed model after offline retraining with newly collected data.

• The computational resources on an embedded system may well satisfy the demand of an inference task. An inference task generally uses input data item by item to perform computations according to its trained model and finally calculate the inference result. Since the input data is used item by item and could be dynamically supplied to the inference task as streaming data with a negligible memory footprint, the memory consumption of such an inference task depends on the size of the model, which is typically on the order of megabytes or gigabytes [2]. And the computation per item of input data is limited to the arithmetic of optimized numerical libraries such as Lapack, Atlas, etc. Its execution time will be in the order of milliseconds or seconds, which allows deployment on an embedded system for an inference task, but not for a training task.

2.2 Unsupervised Learning

There are two primary differences between supervised and unsupervised learning, which make unsupervised learning inappropriate on an embedded system especially for streaming data.

• Unlike supervised learning, unsupervised learning does not use a pre-trained model but follows predefined algorithm to learn on-the-fly.

• Unsupervised learning is temporally iterative but spatially uniform while supervised learning is spatial iterative but temporal uniform. An unsupervised learning task will use the entire data set for the learning computation iteratively until a satisfactory termination condition is met. But a supervised learning task will use the data item by item and generate inference results without temporal dependence.

Consider $K-means$, a classical unsupervised learning algorithm, which defines how the averaging computation will be conducted on the entire input data set. We do not use any trained model during learning. Instead we use iterative computation to update the averaging values for clusters until we reach a fix point.

As discussed, unsupervised learning requires all input data to reside in the memory, which may be feasible for an embedded system with mega- or gigabytes of memory nowadays, as a typical unsupervised learning task tends to consume that much memory for its input data, except for some unusually large inputs up to terabytes [10]. However, an embedded system may receive input at a faster streaming rate than can be processed, thereby exceeding the embedded system's memory. Consequently, it is not appropriate to place an unsupervised learning task on an embedded system.

In summary, only the inference phase of a supervised learning should be placed on an embedded system especially for streaming analysis. Our following work assumes such ML tasks.

3 MODEL PREPARATION AND EXPERIMENTS

Let us next detail how a supervised ML model is trained and how the experiments are conducted.

3.1 The Artificial Neural Network (ANN) Model based on Practical Data

Let us consider a practical industrial problem, ANN technology with its data is utilized to solve a given problem.

With increasing deployment of green power generation source in the power industry, such as solar and wind power, intelligent power control technologies have become important to control power conversion, flow and usage efficiency as well as economics. One of the practical problems solar power management utilities are facing is the prediction of generation capabilities of solar power systems over future time spans. High accuracy of the prediction results is imperative for the cooperation of different power generation and storage devices to ensure peak power control and optimal overall power management cost. ANNs is a mature ML technology that has been applied in industry [7]. Since our research is based on industrial collaboration, we first target ANNs on embedded system to assess their real time suitability.

With our partner, ABB Inc., we have access to field data, which is collected from ABB's $UNO\_2.0\_2.5$ inverter and ABB's weather station $VSN800-14$ from 2014 to 2017 located in Kihei, Hawaii. In general, this data set consists of electrical data from the solar inverter and weather data from the weather station deployed $20km$ away from the solar inverter plant, both of which are collected every five minutes. The electrical data reflects the inverter's DC power generation, and the weather data includes the ambient temperature, the global horizontal irradiance (GHI) and plane of array irradiance (POAI), etc. at the weather station. Since our objective is to predict the inverter's DC power output based on the weather station data, we can define our ML problem as training an ANN model, the output of which is the UNO inverter's DC power output at time k and the input of which are the weather data attributes in the data set at time 0, where k is an positive integer. Here, we are not focusing on how k, the number of future predictions, will affect the accuracy of the model. So for simplicity, we use $k = 1$ for now (but could increase k later).

In order to obtain an ANN model, we first preprocess the data set through splitting and standardization. Since our problem does not have higher dimensional data attributes and we want to preserve the correlation to the input weather data, we simply use all input data attributes, which, in total, amounts to four attributes in our ANN training.

We first train and obtain our ANN model via the Python Keras library [4] with the Tensorflow backend [1]. Our problem is a prediction problem, so we choose Mean Square Error (MSE) as the

evaluation metric for our model. We conduct groups of training experiments to select the best trained model for our assessment of real-time performance suitability (i.e., timing predictability). Our experiments are tuning the number of neuron layers, the number of neurons on each layer, the selection of activation functions on each layer and the selection of training algorithms. Each training experiment iterates over the training data set 30 times to obtain a trained model, which takes about 3 hours on a machine with an X86 64bit Intel i3 processor under Linux Ubuntu 17. After tuning, we pick the best trained ANN model, which has 3 layers of neurons with 64 neurons per layer and the activation functions "Linear", "Rectified Linear Unit (relu)", and "Linear" for each respective layer as trained by the "Stochastic Gradient Descent (sgd)" algorithm. The evaluation metric shows an MSE 0.003 in the final training step. Since the objective is not to find the single optimal ANN model, we stop training when the model has a sufficiently low MSE suitable for inference with high accuracy in terms of variance, namely 0.891.

3.2 Keras vs. Caffe

We obtain the trained model from training with the Keras library. In order to make our model training process reproducible, we control all random variables.

- We eliminate any random number generators except for the random number generator of the Keras Tensorflow backend.
- We set the seed of the pseudo random number generator for the Keras Tensorflow backend to a known constant.
- We disable data shuffling for each epoch of model training.

In order to fairly compare the real time suitability of ML between Keras and Caffe, we use Microsoft's MMdnn tool [3] to convert our ANN model trained with Keras into a model suitable for inference with Caffe.

We conduct experiments to record the execution times of the ML inference task and compare them for Keras and Caffe, both deployed on a Raspberry Pi 3B embedded system. The inference APIs subject to comparison are "model.predict" in Keras, with an option to provide input data item by item, and "net.forward" in Caffe. This ensures a fair comparison for the prediction of each data item. We time the inference APIs using the same numeric value of data inputs to establish upper bounds on execution time bound without skew due to input data variance.

In order to increase real-time predictability on the embedded system, we do not rely on the existing preemptive scheduler of the Rasbian OS on the Raspberry Pi but patch it to become fully preemptive in the Linux kernel [9].

We conducted 10 repeated experiments with 10, 000 execution time samples each, one per prediction on the streamed input data. Fig. 1 compares the average execution time (y-axis) and its standard deviation (error bars) per experiment for the inference API calls between Keras and two Caffe versions, i.e., results indicate the cost of one inference.

3.3 Experiments and Discussion

Figure 1 and Table 1 show that the average execution time of Keras's inference phase is about 4 times slower than that of the original Caffe code basis. However, the standard deviation of the execution time, which is directly related to the upper bound on execution time predictability for the original Caffe code distribution, varies

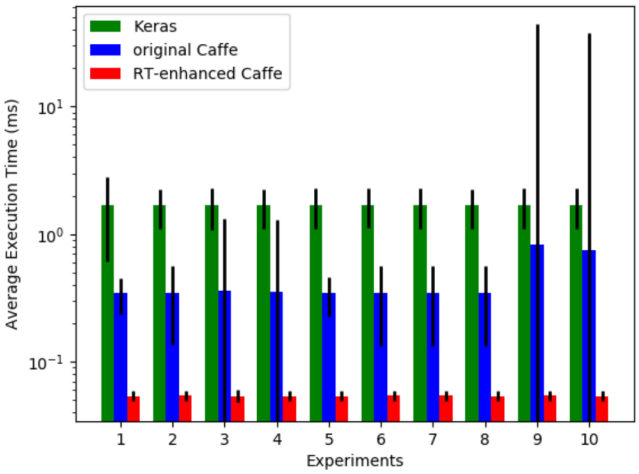

Figure 1: Comparison between Keras, original Caffe and RT-enhanced Caffe

significantly, i.e., it is occasionally two orders of magnitude larger than and otherwise 4 times smaller than that of Keras. This somewhat surprising result (see following discussion) shows that Keras outperforms the original Caffe code in performance and real-time predictability for the ML task of inferencing.

4 CAFFE VS. REAL-TIME ENHANCED CAFFE

4.1 Inspection and Adaptation Of Caffe

Caffe is an open-source ML library written in C++. In contrast to the Python interpreter-based Keras library, the inference code of Caffe is written in C++, compiled into native code, which results in tighter upper bounds of execution time. Another significant advantage of Caffe over Keras is that it utilizes less memory than Keras and does not dynamically allocate/free any of it. Python's background garbage collector does not provide fine-grained real-time control and often perturbs the predictability of execution time of the ML tasks under Keras. The same is true for Python's reliance of an interpreter, which not only adds overhead for execution but also reduces predictability. (Notice that Python's libraries, such as numpy, often make calls to lower-level C or Cuda libraries for CPUs and GPUs, respectively, which results in better and more predictable performance on higher-end platforms, but not on embedded architectures such as the Raspberry Pi.)

After inspecting Caffe's source code [11], we identified several weaknesses that are the cause of real time predictability. Caffe's API "net.forward" (src/caffe/net.cpp) was isolated as the location from where these effects originated. Inspection all nested function invocations from above eventually led to this low-level module, which issues C library calls and system calls. Let us summarize our findings:

- Caffe contains redundant code to generate logging and debugging data emitted by third-party libraries, such as glog and gflag, which are disturbing real time predictability. We could not completely disable the corresponding macros since we have to use the third-party libraries to input the trained model and prepare it for our inference computation. As a result, we could only remove the

diagnostic code inside the inference APIs to reduce the observed interference.

• Caffe stores logging data onto the file system. The file operations add unnecessary I/O load disturbance, which will reduce execution time predictability. The effect exists on PC platforms as well, but it is more pronounced on lower-end embedded edge platforms.

• Caffe supports multi-core execution, which is not necessarily desirable for embedded systems. Even though embedded systems may nowadays have multiple cores for real multi-threading, multi-core execution tends to perturb execution time measurements [?]. Our focus is to isolate ML inference on just one core for measurement purposes. Consequently, we disabled the multi-core functionality of our inference software and bound the process of the inference task to a singular core on the Raspberry Pi.

4.2 Experiments and Discussion

We enhanced the source code of Caffe correspondingly to the above discoveries, this increasing its real-time capabilities, and conduct 10 experiments measuring execution time to compare the performance/predictability between the real-time (RT) enhanced Caffe and the original Caffe code. We investigated 4 metrics (maximum, minimum, average and the standard deviation) of execution time and report 2 of them (due to space limitations) for comparing Keras, the original Caffe code and the RT-enhanced Caffe.

Table 1: Avg. Exec. Time of ML Variants

Run Index	*Avg (ms)*		
	Keras	Original Caffe	RT-Enhanced Caffe
0	1.703	0.344269	**0.0539196**
1	1.670	0.346998	**0.0541247**
2	1.688	0.356934	**0.0539761**
3	1.670	0.352451	**0.053923**
4	1.690	0.34381	**0.0539998**
5	1.700	0.345875	**0.0540965**
6	1.691	0.346298	**0.0540667**
7	1.676	0.34739	**0.0539603**
8	1.688	0.822973	**0.0541702**
9	1.696	0.754966	**0.0539765**

Table 2: StdDev. of Exec. Time for ML Variants

Run Index	*Std Dev*		
	Keras	Original Caffe	RT-Enhanced Caffe
0	1.096	0.107229	**0.00486311**
1	0.562	0.211854	**0.00484784**
2	0.600	0.953092	**0.00603904**
3	0.579	0.943381	**0.00469423**
4	0.587	0.115886	**0.00481081**
5	0.579	0.21243	**0.00480341**
6	0.584	0.212915	**0.00487353**
7	0.577	0.213097	**0.0047033**
8	0.577	43.1206	**0.00477731**
9	0.586	37.1882	**0.00476718**

Figure 1 and Table 1 show that the average execution time of the RT-Enhanced Caffe inference API is about 6 times faster than that of the original Caffe. Moreover, Table 2 shows the standard deviation of the execution time for the RT-Enhanced Caffe is about 25 times smaller than the minimum sample of the original Caffe, and about two orders smaller than that of the Keras, which means the RT-Enhanced Caffe outperforms both original Caffe and Keras in performance and real-time predictability for the ML task of inferencing.

5 CONCLUSION

Overall, our experiments show that an inferencing task of a supervised ML problem can be appropriately solved on an embedded system while the training task of supervised ML problems or an unsupervised ML problems cannot be solved in a timely manner and due to resource constrains, both in computation and storage, on an embedded system.

While the original Caffe library was not suitable for ML problems when high real-time predictability is required, our RT-enhanced Caffe increased both performance and predictability significantly to make inference tasks feasible.

Keras was identified to not be suitable for ML problems of high real-time predictability on embedded systems, due to its interpreted nature as well its dynamic memory management with non real-time background garbage collection, in contrast to RT-Enhanced Caffe.

ACKNOWLEDGEMENT

This work was funded in part by NSF grants 1329780, 1813004.

REFERENCES

[1] Martín Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, New York, NY, USA, 2006. ACM.

[3] Cheng Chen, Jiahao Yao, Ru Zhang, Yuhao Zhou, Tingting Qin, Tong Zhan, and Qianwen Wang. Mmdnn. https://github.com/Microsoft/MMdnn, 2017.

[4] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[5] T.L. Crenshaw, E. Gunter, C.L. Robinson, Lui Sha, and P.R. Kumar. The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures. In *IEEE Real-Time Systems Symposium*, pages 400–412, 2007.

[6] R. Dutta, H. Mueller, and D. Liang. An interactive architecture for industrial scale prediction: Industry 4.0 adaptation of machine learning. In *2018 Annual IEEE International Systems Conference (SysCon)*, pages 1–5, April 2018.

[7] T. Fukuda and T. Shibata. Theory and applications of neural networks for industrial control systems. *IEEE Transactions on Industrial Electronics*, 39(6):472–489, Dec 1992.

[8] Demis Hassabis et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[9] Arnd Heursch, Dirk Grambow, Alexander Horstkotte, and Helmut Rzehak. Steps towards a fully preemptable linux kernel. 2003.

[10] Hyungkeun Jee, Jooyoung Lee, and T DowonHong. High speed bitwise search for digital forensic system. 2012.

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[12] Q. Wang, X. Chai, Y. Wang, D. Liu, M. Chen, Y. Li, X. Liu, and O. Bai. A high data rate, multi-nodes wireless personal-area sensor network for real-time data acquisition and control. In *2017 First International Conference on Electronics Instrumentation Information Systems (EIIS)*, pages 1–5, June 2017.

[13] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, April 2008.

[14] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan. Time-based intrusion dectection in cyber-physical systems. In *International Conference on Cyber-Physical Systems*, pages 109–118, April 2010.

[15] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan. Intrusion dectection for cps real-time controllers. In Siddhartha Kumar Khaitan, James D. McCalley, and Chen Ching Liu, editors, *Cyber Physical Systems Approach to Smart Electric Power Grid*, pages 195–217, January 2015.