PerfNet: Platform-Aware Performance Modeling for Deep Neural Networks

Chuan-Chi Wang
ADLINK Technology Inc.
New Taipei, Taiwan
chuan-chi.wang@adlinktech.com

Ying-Chiao Liao
National Taiwan University
Taipei, Taiwan
d08922004@ntu.edu.tw

Ming-Chang Kao
ADLINK Technology Inc.
New Taipei, Taiwan
fencer.kao@adlinktech.com

Wen-Yew Liang
ADLINK Technology Inc.
New Taipei, Taiwan
william.liang@adlinktech.com

Shih-Hao Hung
National Taiwan University
Taipei, Taiwan
hungsh@csie.ntu.edu.tw

ABSTRACT

The technology of deep learning has grown rapidly and been widely used in the industry. In addition to the accuracy of the deep learning (DL) models, system developers are also interested in comprehending their performance aspects to make sure that the hardware design and the systems deployed to meet the application demands. However, developing a performance model to serve the aforementioned purpose needs to take many issues into account, e.g. the DL model, the runtime software, and the system architecture, which is quite complex. In this work, we propose a multi-layer regression network, called *PerfNet*, to predict the performance of DL models on heterogeneous systems. To train the PerfNet, we develop a tool to collect the performance features and characteristics of DL models on a set of heterogeneous systems, including key hyper-parameters such as loss functions, network shapes, and dataset size, as well as the hardware specifications. Our experiments show that the results of our approach are more accurate than previously published methods. In the case of VGG16 on GTX1080Ti, PerfNet yields a mean absolute percentage error of 20%, while the referenced work constantly overestimates with errors larger than 200%.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Supervised learning by regression**; **Massively parallel and high-performance simulations**.

KEYWORDS

Machine Learning, Benchmark, Performance Prediction, Heterogeneous Systems.

RACS '20, October 13–16, 2020, Gwangju, Republic of Korea

ACM ISBN 978-1-4503-8025-6/20/10…$15.00
https://doi.org/10.1145/3400286.3418245

ACM Reference Format:
Chuan-Chi Wang, Ying-Chiao Liao, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. 2020. PerfNet: Platform-Aware Performance Modeling for Deep Neural Networks. In *International Conference on Research in Adaptive and Convergent Systems (RACS '20), October 13–16, 2020, Gwangju, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3400286.3418245

1 INTRODUCTION

Deep learning (DL) has been getting flourishing in recent years due to image classification[6, 8, 10, 13, 14]. DL models with convolution layers replace traditional feature extraction algorithms and are capable of outperforming general machine learning models in terms of accuracy for image classification problems with large training data. The accuracy of a DL model depends heavily on the architecture of the neural network and the hyper-parameters. Large neural network architectures tend to improve the accuracy, but at the cost of computing power. Meanwhile, instead of using CPU to perform the computation in a DL model, many DL applications are running on heterogeneous systems which utilize graphics processing unit (GPU) or deep learning accelerators (DLA) to carry out the DL computations, which significantly reduces the elapsed time and/or throughput of DL tasks.

The combinations of DL models and hardware architectures represent a design challenge for system developers in terms of performance estimation and design trade-offs. Particularly in the case of embedded systems, in addition to the accuracy of the DL models, performance and costs are very important, but it would be tedious and costly to acquire all the available hardware systems and establish the environment on each system to evaluate the performance of desired DL models, which is impractical or impossible. Thus, a performance prediction method which can give accurate estimates on the performance characteristics of a DL model/hardware combination without the actual hardware would significantly increase the productivity of system developers and support hardware/software co-design projects.

Instead of actual performance measurement, we would like to establish a performance prediction scheme that uses performance model to predict the performance of interested hardware/software combinations with high-level hardware/software specifications, which simplifies the tedious performance profiling works and does not require physical preparation of all the execution environments.

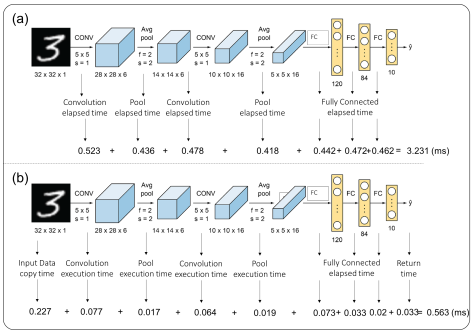

Figure 1: The total elapsed time of LeNet which is run on NVIDIA-1080ti. (a) previous work (b) our work.

Furthermore, it empowers design automation which explores the hardware/software design space. Thus, we proposed *PerfNet* to serve as a performance predictor for DL tasks. Given the information of network model and the target device as input, and *PerfNet* can quickly return the influence time and the resource utilization.

In realizing *PerfNet*, we take a machine learning approach and train a multi-layer regression (MLR) network with performance data collected from existing DL systems. While the previous works [7, 12] also attempted to predict the influence time of neural networks, their models are too simplified and cause excessive prediction errors. In addition to proposing a better prediction model with an improved loss function, *PerfNet* considers the inter-processor communications between CPU and GPU, as well as the overlapped execution time between consecutive neural network layers, and thus delivers more accurate predictions. In summary, the contributions of this paper are as follows:

- A specialized loss function called *MAPLE* is proposed to increase the accuracy of the resulted model.The function is designed based on the unbalanced distribution of input data, as the data are unevenly spaced.
- A new formula for calculating neural network inference time is proposed. As opposed to the previous work [7], the formula delivers far better for heterogeneous systems.
- In addition to prove that the proposed method outperforms the previous works, experimental results are presented to characterize the accuracy of the proposed prediction model with different data-set sizes, neural network model sizes, and hardware platforms.

2 RELATED WORK

Deep learning (DL) models with convolution layers replace traditional feature extraction algorithms and are capable of outperforming general machine learning models in terms of accuracy for many classification problems with large training data. The accuracy of a DL model depends heavily on the architecture of the neural network and the hyper-parameters. In 2015, the depth of some DL models such as ResNet has already reached 200 and the depth keeps growing recently to improve the accuracy, but not considers the cost of computing power.

At the same time, instead of using CPU to perform the computation in a DL model, many DL applications are running on heterogeneous systems which utilize graphics processing unit (GPU) or deep learning accelerators (DLA's) to carry out the DL computations, which significantly reduces the elapsed time and/or throughput of DL tasks. However, the resulted elapsed time and throughput may depend heavily on the architecture of such special purpose processors and cannot be estimated simply based on the number of computing operations generated by a DL model. While certain DL benchmark tools, e.g. MLPerf [2] and dawnbench [5], provide basic performance metrics such as throughput (operations/sec) and latency for performing an inference task, they cannot be used to accurately predict the performance of any DL tasks, as they only cover a small design space of neural network models, whose characteristics may be very different from the DL tasks in question.

To predict the inference time of DL models, Qi et. al. published an open-source tool called PALEO in ICLR 2017 [12], which proposed a way to construct an analytical performance model. PALEO took a small number of representative DL workloads which run operations for a short time on a single GPU to estimate a platform-dependent parameter called *platform percent of peak* (PPP) which captures the average relative inefficiency of the platform compared to peak FLOPS. However, as Daniel et. al. [7] pointed out the analytic model did not consider the architectural details and simply assumed that the execution time of a neural network scaled linearly with the number of operations, which could produce inaccurate results.

Daniel et. al. [7] alternatively proposed an approach by training a DL model for each target platform based on measured data. They considered a neural network as a combination of individual layers, so the essence of the approach was to accurately predict the execution time of each individual layer and then combine the individually predicted times to estimate the total execution time for the entire neural network. Unfortunately, while the method is capable of giving more accurate results than PALEO does for estimating the total execution time on CPU platforms, it does not provide accurate estimates for GPU platforms in our experimental environments. Figure 1 shows the comparison of our work and Daniel's work in heterogeneous system. Figure 1(a) is Daniel's work, they use elapsed time for each layer, which includes the time for CPU to exchange data with the GPU, but in the actual inference, only the first layer and the last layer require such data exchanges. Thus, Daniel's method typically overestimates the total execution time, and Figure 1(b) is our proposed method we overcome this problem by accurate estimate execution time of each layer and additionally considering the system architecture like input/output data copy time. As a result, we improve the prediction accuracy.

To profile the compute time in a heterogeneous computing system and the data exchange time between CPU and DLA, we have used several performance analysis tools such as *SOFA*[11], *FAST*[9], *TensorFlow Profiler*[4], etc. We use SOFA and FAST to acquire coarse-grained performance profile at the system level, and the fine-grained details of neural network operations are left to the TensorFlow Profiler. The profile information includes not only basic CPU and GPU utilization but also collects function call traces to reveal the workflow and break down the execution time for a DL task. For example, through the parser and profiler of Tensorflow, the execution time required by memory copies between CPU and GPU, the time spent

in transforming and replicating codes, and the time used to execute an operation in CPU or GPU can be extracted for us to build detailed performance prediction models.

3 METHODOLOGY

Instead of utilizing CPU resources, DL applications often rely on specialized accelerators (e.g. GPU, FPGA, and DLA) and parallel processing to improve the performance and efficiency, and thus increase the difficulties in analyzing the execution time of the applications. To predict the performance of an accelerator under a given deep learning workload, we need to model the performance for both the CPU and the accelerator, as well as the interactions between them. In addition, we fine-tune a loss function and the architecture of our predictors to improve the accuracy of the prediction models since the training data set is unbalanced. Section 3.1 describes the workflow of the proposed prediction method for homogeneous and heterogeneous platforms. Section 3.2 discusses the specialized loss function. Section 3.3 presents the architectural design of our prediction model, which is based on multi-layer regressions.

3.1 Modeling the Workflow

According to the architecture of the target platform, we illustrate the workflow of our prediction method in Figure 2, which shows the major execution steps to execute a DL inference task on a homogeneous computing system and a heterogeneous computing system. For homogeneous computing systems, Figure 2(a) shows the case of a homogeneous system where only CPU cores are available to initialize the model (*Model Initialization*), perform deep learning operations (*ExecuteOps*), and return the results (*Return Value*) to the application. Since the three steps are executed in serial, the elapsed time is simply the sum of (1) The model initialization time T_{init}, (2) The execution time for the model T_{exe}, and (3) the time to return the results T_{retVal}. The model initialization time depends on the size of the model, the input data, the runtime environment, and the CPU architecture. The execution time mainly depends on the neural architecture and the CPU architecture. Thus, the inference latency for a single batch DL task running on a homogeneous computing system, T_{single_batch}, can be expressed in Equation 1 below:

$$T_{single\_batch}(homogeneous) = \sum_{i=0}^{k} T_{init_i} + T_{exe_i} + T_{retVal_i} \quad (1)$$

where k represents the size of the DL model. In real case, T_{init} and T_{retVal} can be ignored as it only takes a few machine cycles. There we can only measure execution time of each layer for convenience with TensorFlow Profiler.

For heterogeneous computing systems, the workflow is more complicated as it contains the interactions between the CPU process and the DLA. A case with GPU as the DLA is shown in Figure 2(b), where the deep learning operations (*ExecuteOps*) are executed on the GPU after the model (*Model Initialization*) and data (*MemCopy H2D*) are transferred and translated to the GPU (*Replica Code*). Finally, the execution results are transferred form the GPU back to the CPU with *MemCopy D2H* and returned to the calling application via *Return Value*. Through profiling tools, we observed that one GPU process does the *Replica Code* step layer by layer, and another GPU process asynchronously performs the *ExecuteOps* step in a pipeline-fashion whenever a layer is replicated. Thus, we define (1) T_{pre} as the time interval before GPU execution (`ExecuteOps`) as preprocessing time, (2) T_{exe} as the execution time for the model in GPU, and (3) T_{post} to represent the total memory copying time from the GPU device to host and the return value time, i.e. $T_{post} = T_{memCopD2H} + T_{retVal}$. The inference latency for a single batch DL task running on a heterogeneous computing system, T_{single_batch}, can be expressed in Equation 2 below:

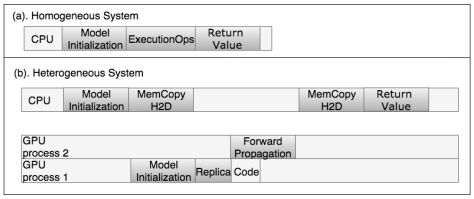

Figure 2: The high-level view of the TensorFlow workflow (a) homogeneous system (b) heterogeneous system.

$$T_{single\_batch}(heterogeneous) = T_{pre_0} + \sum_{i=0}^{k} T_{exe_i} + T_{post_m} \quad (2)$$

By splitting the elapsed time into these three time components for a heterogeneous computing system and modeling them separately, plus the consideration of layer-based pipeline execution, our prediction model delivers more accurate performance estimates than the previously proposed methods.

3.2 Fine-Tuning the Loss Function

Since the training data set in our case has unbalanced distribution and contains some noise from measurement errors, we fine-tune the loss function of our prediction model in the training process, as shown below:

$$MAPLE = \frac{1}{n} \sum_{i=0}^{n} \left| \frac{log(1+\hat{y}_i) - log(1+y_i)}{log(1+y_i)} \right| \quad (3)$$

Instead of using a commonly used lost function, such as *mean squared logarithm error* chosen in [7], we propose to use *mean absolute percentage logarithmic error*, or *MAPLE* for short. The idea came from *mean absolute error* (MAE), which can increase the resistance against the outliers, where the median of all observations is obtained rather than the mean of all target values. We further add the logarithm function of the target at the denominator of our loss function, which allows either long or short elapsed time data to update the balanced weights. Otherwise, the mean absolute percentage error may decrease greatly after every gradient descent.

3.3 Multi-layer Regression Model

To predict the execution time for the time intervals for each layer, i.e. the sum of T_{init_i}, T_{exe_i} and T_{retVal_i}, and T_{pre_0}, T_{exe_i} and T_{post_k}, we train a multi-layer regression model with measurement data and features extracted from the target platform. We fine-tune a multi-layer regression model for our work named *PerfNet*. *PerfNet* is consists of four dense layers with 32, 64, 128, 256 output size respectively, and connect with the dropout layer. To achieve better accuracy, we also fine-tune the hyper-parameters by changing

epoch to 1000 and then divided by 2 every 400 epochs. Finally, the initial learning rate is still set to 0.1 and the batch size is 128. Details of the training process are discussed in the next section.

4 TRAINING THE PREDICTION MODELS

In this section, we describe the process for training the prediction models mentioned in the previous section, including the method of data collection and feature extraction. In this work, we develop a benchmark tool for the user to collect sample data automatically by performing a series of microbenchmarks on the platform of interest. The features extracted by the microbenchmark forms the dataset and are used to train the multi-layer regression models.

4.1 Features

Our benchmark tool samples three major types of neural network layers, including convolution, pooling, and dense layers, with variations of configurations and collects the elapsed time for each configuration. While it is possible to extend this research work to cover the other types of layers, it is in the scope of this paper. Table 1 lists the features that can be varied in our benchmark tool and the range for each feature in three types of layers. Note that the time required for data collection and training are proportional to the number of microbenchmarks, so it is important to design the benchmark tool to efficient cover the application domain with sufficient, effective data samples.

4.2 Data Collection

As mentioned in Section 3, the elapsed time can be broken into several components, and our benchmark tool is capable of extracting these time components by profiling the execution time on the CPU and the GPU, as well as the communication time between the processors in a heterogeneous computing system. In this paper, our case studies use the TensorFlow framework , but the same method can be applied to other DL frameworks, such as Pytorch, Caffee, etc. The hyper-parameters of each layer are randomly generated with uniform distribution to form a microbenchmark. Each microbenchmark is pre-executed 5 times to warm-up the caches before the actual measurement starts. The actual measurement utilizes the API's provided by TensorFlow to extract the execution time intervals. The following describes the measurement process for each type of neural network layer in details:

- **Convolution Layers:** For evaluating the convolution layers, we use the `tensorflow.layers.conv2d` API provided by TensorFlow to create microbenchmarks with various settings, as shown in Table 1. Since large *Batch Size* would exhausts the memory and cause the benchmark to crash, particularly on memory-constrained embedded systems, we chose to limit the *Batch Size* between 1 and 64 to increase the portability of our tool, but the user can manually override the range settings. On the other hand, although *Matrix Size* is 32 for CIFAR-10 and 224 for ImageNet in commonly used, we choose the range from 1 to 512 to cover a large design space. The rest of the range settings are based on our survey of neural network architectures that are commonly used.
- **Pooling Layers:** To generate pooling workloads, we use the `tensorflow.layers.max_pooling2d` API. The pooling layer is usually connected behind the convolution layer, so the selection of the *Pooling Size* is the same as the `Kernel Size` of the convolution layer. The other parameters are of the same ranges and reasons as the convolution layer.
- **Dense Layers:** To benchmark dense layers, we use the `tensorflow.layers.dense` API to generate the workloads. As shown in Table 1, The range of *Dimension Input* or *Dimension Output* can be set between 1 and 4096. The user may override the settings to measure larger dense layers if needed, but such a large dense layer is seldom used in practice. The other parameters resemble the settings for the convolutional and pooling layers.

4.3 Training the Prediction Models

There are out of $7.33 * 10^{14}$, $7.33 * 10^{10}$, and $2.14 * 10^{9}$ possibilities combinations of convolution, pooling, dense layers, respectively. The case studies in this paper samples up to 100,000 randomly chosen combinations and split them into 80,000 as the training datasets (80%) and the remains as the test datasets (20%). Depending on the speed of the target platform and the noise level, it can take more than two weeks to collect 100,000 effective data samples, as every sample is actually an average of several measurements and our tool checks and eliminates obvious measurement errors, and takes approximately 1 hour on a typical workstation with one NVIDIA GTX1080Ti card.

5 PERFORMANCE EVALUATION

We conduct several case studies with experiments to evaluate the impact of different loss functions, different numbers of data, and the multiple layer size of the training models. We set up a homogeneous computing system with an Intel i3-6100TE dual-core processor with 16 GB main memory and a heterogeneous computing system by adding a NVIDIA GeForce GTX 1080Ti (with 11 GB GDDR memory) card. To evaluate the proposed mechanism, 4 performance metrics are set to check the goodness of fit for each model, including mean absolute error (MAE), mean absolute percentage error (MAPE), root mean squared error (RMSE), and coefficient of determination (R^2).

5.1 Effects of the Fine-Tuned Loss Function

The convolution layer is the most time-consuming part of the entire model in a common situation. As a result, all hyper-parameters are tuned to achieve the best accuracy and then applied to the other layers. As shown in Table 2, the results based on the MAPLE loss function are better than those using other loss functions for all devices. The MAPE number is 7.325 % and R^2 is 0.993 which indicates that our model using MAPLE is better than any others. Meanwhile, to answer why using deep learning rather than traditional polynomial regression, we also train it with polynomial number equaling to 4. R^2 is above 0.980 which is still fine. But MAPE is abnormally high and lots of prediction results are negative numbers which are unacceptable. So, It is persuasively enough evidence that MAPLE is the better loss function for 4 metrics.

5.2 Varying the Sizes of Training Datasets

To realize the impact of accuracy on different dataset sizes. We also focus on convolution layer data and fix the testing dataset size

Table 1: Description of the features.

TensorFlow API	conv2d	max_pooling2d	dense	description
Batch Size	1-64	1-64	1-64	The number of parallel processed.
Matrix Size	1-512	1-512	-	The size of the input data.
Kernel Size	1-7	-	-	The size of the filter applied to the image.
Channel Input	1-9999	1-9999	-	The number of channels in the input data.
Channel Output	1-9999	-	-	The number of channels in the output data.
Strides	1-4	1-4	-	The factor to downscale the image with convolution and pooling kernels.
Padding	0-1	0-1	-	The number for preserving the original size of the image by the filter. 0: Valid, 1: Same.
Activate Function	0-1	0-1	0-1	Activation function to use. 0: Not use activate function. 1: Relu activate function.
Bias	0-1	0-1	-	Boolean, whether the layer uses a bias vector.
Dimension Input	-	-	1-4096	The number of outputs from the previous layer.
Dimension Output	-	-	1-4096	The number of outputs of the layer.
Pooling Size	-	1-7	-	The size of the pooling windows, factors by downscale the input.

Table 2: Comparison of Loss function test

Device	NN Shape	Loss Function	MAE(ms)	MAPE(%)	RMSE(ms)	R^2
1080ti	Related Work [7]	MSLE	0.603	8.204%	1.306	0.992
	PerfNet	MSLE	0.616	8.396%	1.323	0.992
	PerfNet	MAPLE	0.538	7.325%	1.233	0.993
	Polynomial Regression	MSE	1.333	43.963%	405.308	0.980
i3	Related Work [7]	MSLE	14.446	12.776%	43.978	0.966
	PerfNet	MSLE	13.994	12.498%	42.657	0.968
	PerfNet	MAPLE	12.335	10.886%	37.932	0.975
	Polynomial Regression	MSE	41.550	305.889%	7960.538	0.909

Table 3: Different datasize test on PerNet and MAPLE loss

Data size	MAE (ms)	MAPE (%)	RMSE (ms)	R^2
5000	1.204	16.479%	2.716	0.967
10000	0.824	12.351%	1.746	0.986
20000	0.763	10.135%	1.666	0.987
40000	0.653	8.615%	1.414	0.991
80000	0.538	7.325%	1.233	0.993

as 20,000 and cumulatively increasing training set size from 5,000 to 80,000. All the performance metrics would get better while the dataset becomes bigger. Because collecting and verifying data need a great amount of cost, we trade-off for benchmarking 100,000 data as our dataset. (see Table 3).

5.3 Prediction Accuracy for Individual Layers

In this section, we first fine-tune PerfNet by varying its layer size from 1 to 6. As shown in Figure 3, the experimental results indicate that PerfNet give better prediction accuracy when the layer size is set to 4, i.e. *PerfNet 4*, as a larger layer size may cause overfitting. Then, we use PerfNet 4 for the following experiments to evaluate the prediction accuracy for individual layers. Table 4 lists the MAE, MAPE, and RMSE of the predictions from PerfNet for various types of layers on our homogeneous system setup. Among the three types of layers, convolution layers are more difficult to predict than the other pooling and dense layers because of its complexity. The MAPE for convolution layers is 10.9%, and the R^2 is 0.98. For our purpose of supporting system designs, such level of accuracy is fine. Notice that

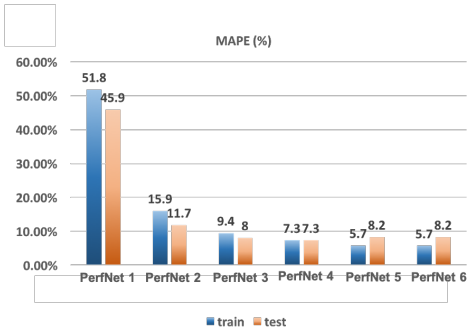

Figure 3: Mean absolute percentage error (MAPE) of PerfNet architectures with different layer sizes

Table 4: Prediction accuracy for the Intel i3-6100TE system.

Device	Layers	MAE (ms)	MAPE (%)	RMSE (ms)	R^2
i3	convolution	12.2	10.9%	37.7	0.980
	pooling	7.6	8.3%	30.2	0.980
	dense	0.08	3.2%	0.12	0.999

the RMSE values for the convolution and pooling layers are quite high, i.e. 37.7 ms and 30.2 ms respectively, which is due to the fact that the CPU in the system could take several seconds to complete a microbenchmark and may be interfered by the other events in the system during the measurement period. For heterogeneous systems, we model the workflow proposed in Section 3.1 and split

Table 5: Predition accuracy for the heterogeneous system.

Device	Layers	Phase	MAPE(%)	RMSE(ms)
1080ti	convolution	preprocess	2.380%	0.236
		execution	14.960%	0.985
		postprocess	3.741%	0.117
	pooling	preprocess	2.045%	0.220
		execution	7.669%	0.373
		postprocess	2.776%	0.097
	dense	preprocess	6.504%	0.037
		execution	4.526%	0.068
		postprocess	13.887%	0.011

each layer into three-phases. Table 5 lists the accuracy of PerfNet for predicting the three phases in various types of layers. Similar to the aforementioned results from our homogeneous system case study, the prediction for convolution layers is not as accurate as the other two types of layers. However, with our 3-phrase approach, the accuracy for the system with a NVIDIA GPU is better than the accuracy for the system without a GPU. The RMSE value for the predicted execution time in a convolution layer is 0.985 ms, which is vastly improved from the RMSE of 2.55 ms produced by the method published in [7]. As the inference time of the DL model is dominated by the execution of the convolution layer, improving the prediction accuracy for convolution layers in critical.

5.4 Prediction Accuracy for Full Networks

In this section, we use *PerfNet* to predict the performance for full deep neural networks using equations derived in Section 3.1 and compare it to a predictor based on the previous work [7]. We chose three popular used deep neural networks for this experiment: LeNet [10], AlexNet [8], and VGG16 [13]. LeNet represents a small neural network, VGG16 represents a large neural network which contains mass operations and parameters, and AlexNet is between them. Figure 4 compares the actual execution time, the predicted execution time from the previous method and our work (PerfNet), for each of these neural networks to complete an inference on a homogeneous (CPU) or heterogeneous (GPU-1080ti) platform. Since the batch size impacts the performance substantially, we vary the batch size from 1 to 32 in this experiment. Figure 4(a) shows the cases on CPU, where both predictors overestimate the execution time for LeNet and AlexNet, but underestimate for VGG16. For the cases on CPU, our work exhibits better prediction accuracy for AlexNet, but loses to the previous work in the case of VGG16. On the other hand, Figure 4(b) shows the strength of our work, as PerfNet delivers far more accurate predictions than the previous work does. We believe that the better accuracy comes mainly from the 3-phase approach that we proposed for heterogeneous systems in this research work.

6 CONCLUSION AND FUTURE WORK

In this work, we proposed a multi-layer regression model with a strong loss function to deliver more accurate predictions than related works. We have shown the workflow of using our proposed method to predict the inference time for full deep learning models, including LeNet, AlexNet, and Vgg16. Our experimental results confirm that our method outperforms the previous work particularly

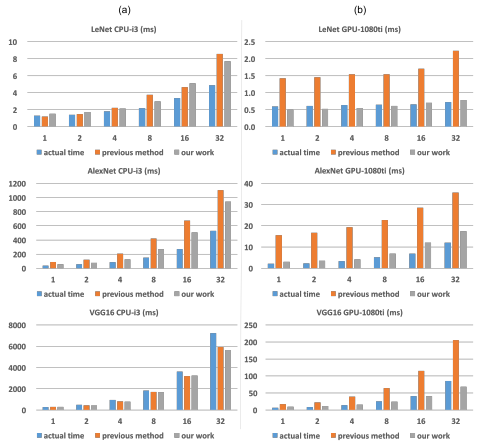

Figure 4: Full model prediction for LeNet, AlexNet and VGG16 on (a) CPU-i3 and (b) GPU-1080ti.

for heterogeneous computing platforms where the previous work ignores the CPU-GPU data exchange time. With the encouraging results, we would like to further improve the accuracy of prediction via feature transformation and noise reduction. Furthermore, we plan to extend our approach to cover more DL acceleration frameworks such as OpenVINO [3] and TensorRT [1], as well as new hardware devices such as Intel's VPU.

REFERENCES

[1] 2014. NVIDIA TensorRT. (2014). https://developer.nvidia.com/tensorrt
[2] 2018. mlperf. (2018). https://mlperf.org/
[3] 2018. OpenVINO. (2018). https://software.intel.com/en-us/openvino-toolkit
[4] 2020. Tensorflow Profiler. (2020). https://www.tensorflow.org/tensorboard
[5] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. *Training* 100, 101 (2017), 102.
[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*.
[7] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. 2018. Predicting the computational cost of deep learning models. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 3873–3882.
[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[9] Cheng-Kung Lai, Chih-Wei Yeh, Chia-Heng Tu, and Shih-Hao Hung. 2017. Fast profiling framework and race detection for heterogeneous system. *Journal of Systems Architecture* 81 (2017), 83–91.
[10] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[11] Cheng-Yueh Liu, Po-Yao Huang, Chia-Heng Tu, and Shih-Hao Hung. 2018. A Fast and Scalable Cluster Simulator for Network Performance Projection of HPC Applications. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 970–977.
[12] Hang Qi, Evan R Sparks, and Ameet Talwalkar. 2017. Paleo: A performance model for deep neural networks. (2017).
[13] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[14] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.