BayesTuner: Leveraging Bayesian Optimization For DNN Inference Configuration Selection

Seyed Morteza Nabavinejad and Sherief Reda, *Senior Member, IEEE*

Abstract—Deep learning sits at the core of many applications and products deployed on large-scale infrastructures such as data centers. Since the power consumption of data centers contributes significantly to operational costs and carbon footprint, it is essential to improve their power efficiency. To this end, both the hardware platform and application should be configured properly. However, identifying the best configuration automatically for a wide range of available options with affordable search cost is challenging (e.g., DNN batch size, number of cores, and amount of memory allocated to the application). Employing an exhaustive approach to test all the possible configurations is unfeasible. To tackle this challenge, we introduce *BayesTuner* that employs Bayesian Optimization to estimate the performance model of deep neural network inference applications under different configurations with a few test runs. Having these models, *BayesTuner* is able to differentiate the optimal or near-optimal configurations from the rest of options. Using a realistic setup with various DNNs, we show how efficiently BayesTuner can explore the huge state space of possible control configurations, and minimize the power consumption of the system, while meeting the throughput constraint of different DNNs.

Index Terms—Deep Neural Network, Power, Throughput, Bayesian Optimization

1 Introduction

Various systems are designed and implemented for executing DNN inference. While these systems are based on different hardware platforms, the deployment of DNN inference applications on them has the similar structure: placing the pre-trained model on the system, allocating a specific amount of resources (e.g., CPU, RAM), and adjusting the DNN-side control knobs such as batch size, which defines how many inputs should be processed at a time in the form of a batch.

Finding and selecting the right configuration (e.g., batch size, number of cores, allocated memory) is essential for improving performance and power efficiency of the system. Quality of service and customer satisfaction have a direct relationship with the performance. Moreover, power consumption has a significant share in the operational costs of data centers and determines their carbon footprint [1]. The impact of the configuration is especially important for recurrent jobs that deploy similar DNN inference applications periodically on the infrastructure, or the long-running jobs that process a large input dataset.

We conclude that it is challenging to find the proper configuration for various objectives, e.g., maximizing the throughput or minimizing the power consumption, because of the complexity of building performance models in the presence of various control knobs. Since the control knobs have complex relationship with performance and power, it is hard to use common methods to accurately model this relationship. Moreover, using an exhaustive search method to find the right configuration imposes significant overhead, and hence, is unfeasible. For instance, the size of state space (all the possible configurations) in this work is 7168 configurations. In a realistic setup, running test samples to understand the behavior of application regarding different control knobs is very expensive, and hence, the number of test samples is very limited.

- Seyed Morteza Nabavinejad is with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. E-mail: nabavinejad@ipm.ir
- Sherief Reda is with the School of Engineering, Brown University, Providence, RI. S. Reda work is partially supported by NSF grant 1814920 and DoD ARO grant W911NF-19-1-0484. E-mail: sherief_reda@brown.edu

To address the aforementioned challenges, we present *BayesTuner*, a low-overhead adaptive approach for diverse set of recurring and long-running DNN inference applications that can find the optimal or near-optimal configuration, such that the power consumption of the hardware platform is minimized, while the throughput of the application is not less than a predefined constraint. For the CPU-based hardware platforms, the configuration includes the number of cores and amount of memory allocated to the job. We also consider batch size, as batching is widely used in previous works for increasing the throughput of DNN inference [2], [3].

The key idea of *BayesTuner* is identifying near-optimal configurations. Therefore, it needs a performance model with enough accuracy that can distinguish the near-optimal configurations from the rest of configurations. This feature helps *BayesTuner* to achieve low overhead as finding the near-optimal configuration needs fewer sample runs. To build this accurate enough performance model, *BayesTuner* leverages Bayesian Optimization (BO). BO is able to optimize black-box functions, and hence, it does not need application-specific insights such as the architecture of DNN (number of layers, type of layer, etc.) or low-level profiling of hardware platform (cache, memory, etc). It is especially important as there is a trend toward application-agnostic optimization approaches in data centers due to security and privacy concerns [1]. We use 8 DNNs from different domains and a CPU-based hardware platform to evaluate the efficacy of *BayesTuner*. The results indicate that *BayesTuner* can yield up to 25% and 39% improvement in power consumption in the presence of a throughput constraint, compared with two other approaches that do not leverage BO. It also can find solutions as close as 1.4% to optimal ones on average, when compared with Exhaustive Search.

1.1 Motivation

We select three DNNs from different domains (DeepSpeech from speech recognition, DeePVS from video saliency, and Inception from image classification) to study the impact of a set of diverse configurations on their power and throughput. We consider the number of cores allocated to the DNN and

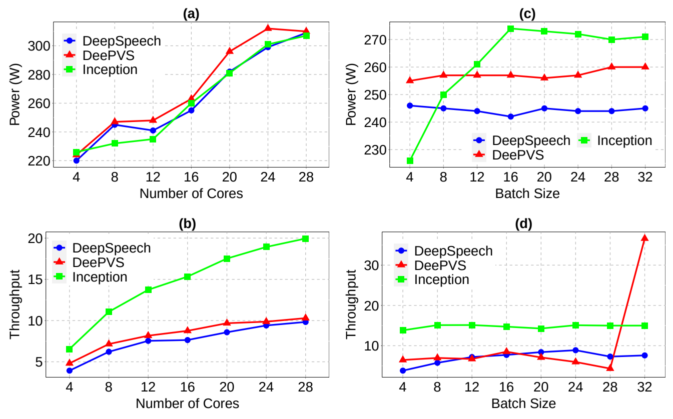

Fig. 1. Impact of number of cores and batch size on the power and throughput of three selected DNNs.

the Batch Size (BS) as control knobs. We consider a baseline configuration and change each of the control knobs separately. When changing the BS, we consider the number of cores a fixed value (14 cores using cgroups feature of Linux), and when we change the number of cores, we consider BS as fixed value of 16. The power consumption and throughput of DNNs is presented in Fig. 1. The throughput for DeepSpeech is the number of audio files to process per second, for DeePVS is the number of video frames per second, and for Inception is the number of images to classify per second.

We observe that a proper configuration can help significantly reduce the power consumption to achieve a certain throughput. For example, in Fig. 1.(c) and (d), both BS = 4 and and BS = 20 can reach almost the same throughput, 13.8 and 14.2 respectively. But one with power consumption of 86 W (BS = 4) and the other with power consumption of 115 W (BS = 20). Therefore, wrong configuration selection can waste 33% power, without achieving higher throughput. It is especially important for recurring jobs, where similar workloads are repeated periodically, or long-running jobs where a huge amount of data should be processed. Our approach is suitable for these kind of jobs as the resource and time overhead of searching for the near-optimal configurations can be amortized over the course of time.

2 BayesTuner

Our knowledge of the vast state space of configurations is limited to a few ones that we can explore in test runs, which are very costly. Hence, instead of trying to build an accurate performance model for each DNN to find the best configuration, we aim to estimate a model that is accurate enough to help us identify the optimal or near-optimal configurations.

2.1 Problem Statement

For a given DNN and a hardware platform, find the best configuration (i.e. the combination of control knobs provided by DNN and platform) to minimize the power consumption subject to a performance requirement. In this work, we consider the throughput as the performance metric. The throughput T(BS, core, memory) and power consumption P(BS, core, memory) depend on the DNN application and hardware platform control knobs, i.e., batch size, number of cores, and amount of memory. Our ultimate objective is to find the best configuration that minimizes the power consumption, while having throughput greater than or equal to a predefined throughput threshold T_{th}:

$$\begin{aligned} &Minimize \quad P(BS, core, memory) \\ &S.t. \quad T(BS, core, memory) \geq T_{th} \end{aligned} \tag{1}$$

By testing all the configurations, we can have T(BS,core,memory) and P(BS,core,memory) for all of them and easily solve Equation (1); but, it is extremely costly. Using BO, *BayesTuner* can find an approximate solution for Equation (1) by testing a much smaller subset of configurations that are selected dynamically, and hence, significantly decrease the search cost.

2.2 Bayesian Optimization Principals

BO models the unknown function, e.g., P(BS, core, memory), with a stochastic process (called Prior function) and tries to estimate it by the help of samples taken at different points of that unknown function. After taking each sample, the estimated model and the confidence interval that shows the difference between the estimated model and the actual model are updated by BO. For selecting the next sample point wisely, BO relies on a pre-defined acquisition function. The acquisition function determines the sample that can yield the highest expected improvement of the estimated model, such that the confidence interval is became narrower. The illustrative example in Fig. 2 shows how BO works.

For the Prior function (i.e., the stochastic process to estimate the objective function and constraint models based on), we use the Gaussian process, as it is a common and accepted option for BO [4]. This choice means that we assume the unknown function(s) is a sample from Gaussian process. The Gaussian process estimates the actual function f with a surrogate model f'. In f', for each input (i.e., configuration) the output is defined by a random variable, instead of an actual value. This random variable tells what is the possible value for function f (i.e., power consumption and throughput in our work) for a certain input configuration. At the beginning, the degree of uncertainty is high, which mean the estimated output for a certain input has a wide confidence interval. As more samples are taken, the uncertainty is decreased and confidence interval becomes tighter, means that the estimated output of input configurations is more accurate. Flexibility of this non-parametric process allows to come close to the actual function by taking enough samples. The number of samples need to be taken to reach to the actual function depends on the similarity of that function with Gaussian process. Closer functions need fewer numbers of samples to be accurately estimated. It is possible to find better prior functions than Gaussian process for specific DNNs by having knowledge about them. However, it renders the generality of that prior function for broader range of DNNs low [5].

For the acquisition function, which determines the next sample point to take, we use Expected Improvement (EI) which selects the sample point that might maximize the expected improvement over the current best result. The EI method is the most popular option over the other possible options and does not require self-tuning [4]. The EI takes into account the estimated model (f') yielded by the Gaussian process up to this point. It also considers the best (lowest in our work) value obtained for objective function until now from the actual samples taken. Then, it examines the remaining configurations in the state space by the estimated model and obtains the objective value for each of them. Then it selects the one that can improve the objective function the most compared with the best value identified until now. This new configuration is selected as the next test sample and is executed to find the real value of objective function for it. Then, this new test sample and its objective value is fed to Gaussian process, along with the previous test samples and their values, to update the estimated

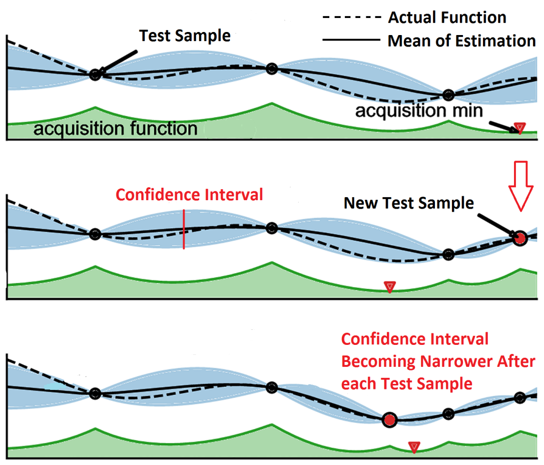

Fig. 2. Illustrative example to show how BO works (adapted with modification from [6]). EI finds the configuration corresponding as global extremum (minimum in this sample) and selects that as the next example.

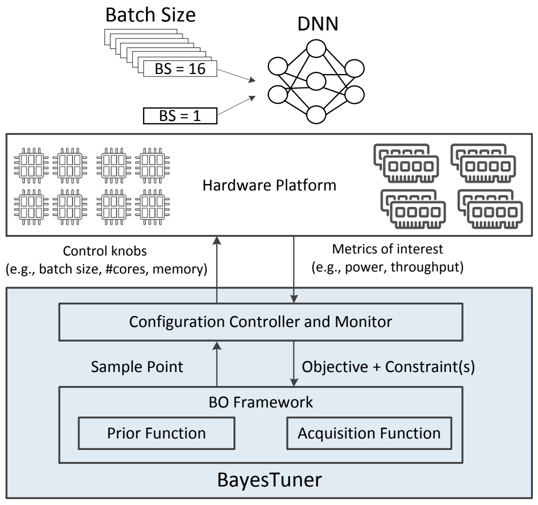

Fig. 3. Overall flow of BayesTuner

model. This loop is repeated until we reach the maximum number of test samples.

2.3 Methodology

The overall flow of *BayesTuner* is shown in Fig. 3. It consists of two modules which we discuss in the following: Configuration Controller and Monitor and BO Framework.

Configuration Controller and Monitor. This module interacts with BO framework, hardware platform, and the application, and orchestrates the entire process of setting the values of control knobs of hardware platform and application, launching test runs for sample points determined by BO framework, as well as monitoring the objective function and constraints for sample points and sending them to the BO framework. The user supplies this module with the DNN model of the inference application, and the desired objective and constraints (e.g., power consumption, throughput, execution time, etc.). For the DNN inference application, this module sets the batch size based on the value received from BO framework. For adjusting the control knobs of hardware platform (e.g., number of cores and amount of memory), it uses control groups (cgroups) feature of Linux. For other hardware platforms with specific features, other parameters and control knobs can be adjusted by this module.

TABLE 1
Specifications of Jobs Used in the Experiments.

DNN	Dataset	Domain	Throughput Constraint
DeepSpeech [8]	Sentiment140 [9]	Speech Recognition	6 audio/sec
DeePVS [10]	LEDOV [10]	Video Saliency	4 frame/sec
TextAnalysis [11]	LibriSpeech [12]	NLP	5000 sentence/sec
PNASNet [13]	Imagenet [14]	Image Classification	4 image/sec
NASNet [15]	Imagenet	Image Classification	3 image/sec
InceptionV3 [16]	Imagenet	Image Classification	20 image/sec
ResNetV2 [17]	Imagenet	Image Classification	10 image/sec
MobileNet [18]	Imagenet	Image Classification	50 image/sec

BO Framework. For Bayesian Optimization framework, we leverage Spearmint [7] which is implemented in Python and supports both the Gaussian process for prior function and EI approach for the acquisition function. For taking a sample, BO submits the specifications of the selected configuration to the Configuration Controller and Monitor module. After completion of test run, BO receives the desired metrics for objective function (power) and constraint (throughput) from the same module.

3 Evaluation

3.1 Experimental Setup

Hardware platform. We run our experiments on a dual-socket Xeon server where each of the E5-2680 v4 Xeon chips has 14 cores running at 2.4 GHz. The server has 128 GB of DDR4 memory. Ubuntu 16.04 with kernel 4.4 is installed on the server with the python 2.7, CUDA 11.0, and TensorFlow 1.15.

DNN jobs. To show the adaptive nature of our approach, we use DNNs from different domains. The selected DNNs cover a wide range of applications, as well as DNN types: from CNNs to RNNs, to LSTMs. The jobs used in the experiments are shown in Table 1

Objective and constraint. The objective function is defined as minimizing the power consumption of the DNN inference application under a throughput constraint. By default, we consider a loss throughput constraint for each DNN, so there is more room to explore the state space.

Systems compared. We compare *BayesTuner* with three strategies: 1) *Exhaustive Search*, that tests all the possible configurations to find the best one. 2) *BatchSizer* [2], that uses the DNN control knob (batch size) to manage the power consumption and throughput. It does not consider the hardware platform control knobs, and hence, uses the entire system resources. 3) Simulated Annealing (SA) is a meta-heuristic approach that approximates the global optimum of problems with large state space. It explores the state space by probabilistically deciding to move to a neighbor configuration. SA does not estimate a performance model for objective and constraint and hence, generally converge slower than BO.

State space. The hardware platform, as mentioned, has 28 cores. While the available memory of this server is 128 GBs, our observations show that none of the DNNs consume more than 8 GBs of RAM. Hence, to moderate the size of state space, we consider 8 GBs of RAM, which can be changed by steps of 1 GBs. We use cgroups feature of Linux to manage amount of memory. Considering the batch size for DNNs as a number between 1 to 32, the total number of possible configurations that we can select from is $28 \times 8 \times 32 = 7168$. For comparing *BayesTuner* against SA and BatchSizer, we consider this state space. But, for comparing *BayesTuner* against Exhaustive Search approach, we consider a smaller version of state space where the number of cores and batch size can be set with steps of 4 (e.g, 4, 8, 12) and the amount of memory can be set with

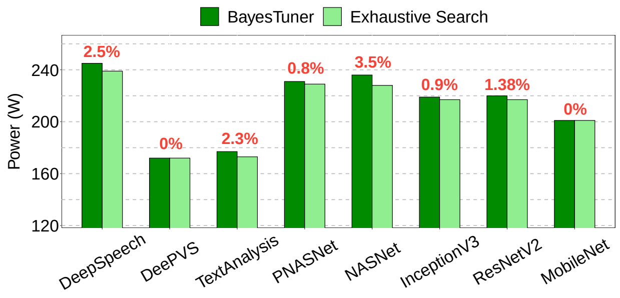

Fig. 4. Comparing the results of BayesTuner against optimal solution for power consumption. The red numbers on top of the bars indicate the difference between BayesTuner and optimal solution. The number of test samples for *BayesTuner* is 20 and for Exhaustive search is 224.

steps of 2 (e.g, 2, 4, 6). In this way, the number of all possible configurations would be $7 \times 4 \times 8 = 224$. The internal controller of the CPUs control Dynamic Voltage Frequency Scaling (DVFS) and we do not apply any changes on it, nor using any control approach.

3.2 Results

***BayesTuner* can find near optimal solutions with less search cost.** *BayesTuner* can find solutions similar to the optimal one. In Fig. 4, the power consumption of optimal solution (achieved by Exhaustive Search) and BayesTuner is depicted for all the DNNs. The average difference between optimal solution and *BayesTuner* is 1.4%. It clearly shows the success of *BayesTuner* at finding optimal and near-optimal solutions with test cost remarkably lower than that of Exhaustive Search. *BayesTuner* only selects 20 test samples from the state space (9% of the configurations), while Exhaustive Search tests all the 224 configurations. The total search time of *BayesTuner* and Exhaustive Search is shown in Table 2. *BayesTuner* can dramatically reduce search time (up to 96% and 88% on average) compared with Exhaustive Search. These results can emphasize the efficacy of using BO to reduce the search cost, while achieving optimal or near-optimal solutions. Furthermore, we have estimated the sampling time of Exhaustive Search for the original state space with 7168 configurations (multiplying the sampling time of each job in small state space by $\frac{7168}{224}$). Comparing them with sampling time of *BayesTuner* clearly shows how using BO becomes more prominent as the state space expands.

***BayesTuner* can significantly improve the objective function with the same search budget as SA.** The power consumption results of *BayesTuner*, SA, and BatchSizer for the same number of test samples (20) is depicted in Fig. 5. *BayesTuner* improves the power consumption by up to 25% (7% on average) compared with SA. It emphasizes the power of *BayesTuner* in selecting the sample points wisely, in contrast to SA that can be trapped in local minimums. The acquisition function of our BO framework can successfully guide the sampling process to the right direction, in order to select better sample points and build a more accurate performance model for the application. The slower convergence problem of SA shows itself in the form of higher power consumption in the results, as it needs more test samples to approach the optimal solution.

***BayesTuner* leverages all the control knobs of both hardware platform and the application, and hence, achieves better solutions than the approach that only leverage the application-side control knob.** The BatchSizer only tunes batch size to find a solution. Therefore, it misses the opportunity to explore the entire state space, and hence, its ability to find optimal or near-optimal solutions is seriously degraded. The strength of our approach, however, is that it can leverage the cross-stack control knobs to better navigate the state space and find better solutions. The maximum power consumption improvement of *BayesTuner* over BatchSizer is around 39% and the average improvement is 26%.

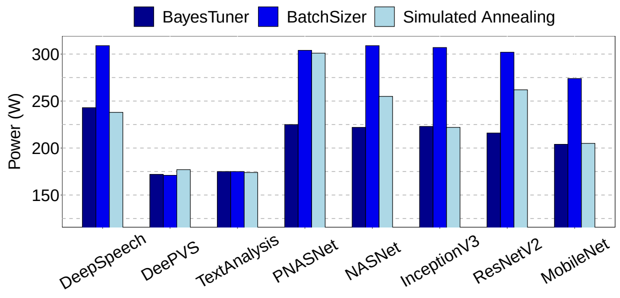

Fig. 5. Comparing the results of BayesTuner against Random Search and BatchSizer with the same number of test samples.

TABLE 2
Sampling Time (seconds) of Different Approaches.

	Small State Space		Large State Space	
	BayesTuner	Exhaustive	BayesTuner	Exhaustive (Estimated)
DeepSpeech	1137	8521	3936	272659
DeePVS	203	6181	663	197778
TextAnalysis	398	4968	861	158966
PNASNet	807	5520	1147	176636
NASNet	758	5249	1234	167972
InceptionV3	664	4327	1026	138475
ResNetV2	404	3036	642	97150
MobileNet	355	2565	546	82076

3.3 Detailed Analysis of BayesTuner

In this section, we explore the behavior of *BayesTuner* in more details. In Table 3, the 20 sample points selected by *BayesTuner* for MobilenetV2 DNN is listed and in Fig. 6, the power and throughput of those points is depicted. The horizontal line in Fig. 6 indicates the throughput constraint of this DNN. *BayesTuner* can find a solution that meets the throughput at test sample 2. However, it tries to find another valid solution, but with less power consumption in the following. In its exploration, it aims to identify the control knob that has more effect on power and throughput, and hence, it tests different values for number of cores, amount of memory, and batch size. at the first few samples, it explores the edges of the state space. Eventually, the sample points selected by BayesTuner are closer to the throughput constraint, because it realizes that to minimize the power consumption, the throughput should be close to the constraint. It stops after reaching the maximum number of samples (20) is reached.

Earlier in Section 1.1, we mentioned that in addition to recurrent jobs, the long-running jobs can also benefit from choosing right configuration. Considering the sampling time of *BayesTuner* shown in Table 2, even jobs that do not belong to the aforementioned categories, but are long enough compared to sampling time, can also benefit from results of *BayesTuner*. The size of sample input directly affects the sampling time. Therefore, choosing small inputs that are able to capture the characteristics of jobs (e.g., power, throughput), can help to reduce the sampling time, and hence, employ *BayesTuner* for a wider range of jobs with shorter runtime. In this case, *BayesTuner* can be even used for scenarios where the QoS (e.g., throughput constraint) of jobs changes dynamically during runtime. The new configuration with respect to new QoS can be found by the help of BO, provided that the job can tolerate

TABLE 3
The specifications of 20 test samples selected by BayesTuner for MobilenetV2 DNN (BS: Batch Size).

Test Sample	# Core	Mem (GB)	BS	Test Sample	# Core	Mem (GB)	BS
1	1	1	1	11	6	8	29
2	14	4	16	12	19	2	1
3	1	8	32	13	18	1	1
4	28	1	4	14	12	1	1
5	9	8	1	15	19	8	1
6	4	1	32	16	5	1	9
7	27	8	1	17	4	1	3
8	28	7	1	18	4	2	1
9	3	2	1	19	22	5	1
10	23	8	32	20	19	8	1

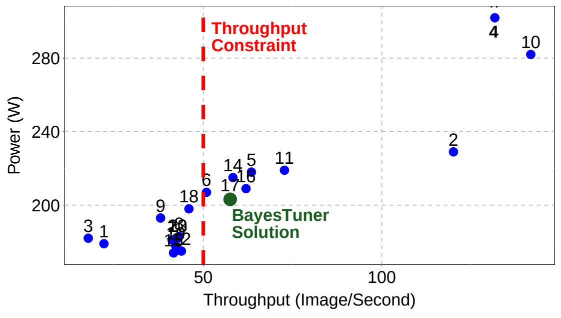

Fig. 6. Detailed behavior of BayesTuner for MobilenetV2 DNN.

the QoS violation for a certain period, until finding the new configuration.

4 Related Work

Improving throughput and power-efficiency of DL systems via fine-tuning application/framework-side control knobs has been studied in previous works. A large body of research uses batch size as a control knob to increase the throughput or improve the power-efficiency, while meeting a certain latency constraint. These works use mechanisms such as binary search [2] or additive-increase-multiplicative-decrease (AIMD) [19] to find the proper batch size. auto-tuning framework parameters (e.g., number of operators to execute in parallel) is the focus of another group of previous works [20], [21]. These works do not consider tuning the hardware platform control knobs, simultaneously with application and framework parameters, to manage throughput/power consumption. Reagen et al. [22] also employ BO for designing a hardware accelerator for training phase of DNNs, in contrast to our work that uses BO for inference phase on CPU-based hardware platform. Bayesian Optimization (BO) has been leveraged to find the proper virtual machine (VM) configuration for big data jobs in Cherrypick [5]. CLITE [23] also employs BO to find the proper configuration for co-locating several latency-critical jobs with background jobs. RAMBO [24] also uses BO to find the Pareto-front of microservices by solving a multi-objective problem. All these approaches only consider the hardware platform control knobs (e.g., number of CPUs, memory bandwidth, etc.) and ignore application-side control knobs. Unlike the prior approaches discussed in this section, *BayesTuner* incorporates both hardware and application control knobs in BO to achieve better results. Simultaneous coordination of both application and hardware control knobs lead to a larger configuration space with more number of configurations, and consequently, cause BO to spend more time exploring it for finding suitable test samples. However, it provides this opportunity to find optimal or near optimal solutions that otherwise would not be possible to obtain due to non inclusion some control knobs.

5 Conclusion

We presented BayesTuner, an automated configuration selection framework for DNN inference applications leveraging Bayesian Optimization. Using a real-world setup with several DNNs and a high-end hardware platform, we showed that BayesTuner can efficiently explore the state space of configurations and find optimal or near optimal solutions that minimize the power consumption, while meeting the throughput constraint. BayesTuner can be extended to cover various types of hardware platforms, such as GPU clusters, in addition to the CPU-based platform used in this paper.

References

[1] K. Kaffes, D. Sbirlea, Y. Lin, D. Lo, and C. Kozyrakis, "Leveraging application classes to save power in highly-utilized data centers," in *SoCC*, 2020, pp. 134–149.
[2] S. M. Nabavinejad, S. Reda, and M. Ebrahimi, "Batchsizer: Power-performance trade-off for dnn inference," in *ASP-DAC*, 2021, pp. 819–824.
[3] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *SC*. IEEE Computer Society, 2020, pp. 972–986.
[4] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *arXiv preprint arXiv:1206.2944*, 2012.
[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI*, 2017, pp. 469–482.
[6] L. L. Grado, M. D. Johnson, and T. I. Netoff, "Bayesian adaptive dual control of deep brain stimulation in a computational model of parkinson's disease," *PLoS computational biology*, vol. 14, no. 12, 2018.
[7] Spearmint. [Online]. Available: https://github.com/HIPS/Spearmint
[8] D. Amodei and et al., "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*. PMLR, 2016, pp. 173–182.
[9] Sentiment140. [Online]. Available: http://help.sentiment140.com/
[10] L. Jiang, M. Xu, T. Liu, M. Qiao, and Z. Wang, "Deepvs: A deep learning based video saliency prediction approach," in *ECCV*, 2018, pp. 602–617.
[11] Y. Kim, "Convolutional neural networks for sentence classification," *CoRR*, vol. abs/1408.5882, 2014. [Online]. Available: http://arxiv.org/abs/1408.5882
[12] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *ICASSP*. IEEE, 2015, pp. 5206–5210.
[13] C. Liu and et al., "Progressive neural architecture search," in *ECCV*, 2018, pp. 19–34.
[14] J. Deng and et al., "Imagenet: A large-scale hierarchical image database," in *CVPR*. Ieee, 2009, pp. 248–255.
[15] B. Zoph and et al., "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018, pp. 8697–8710.
[16] C. Szegedy and et al., "Rethinking the inception architecture for computer vision," in *CVPR*, 2016, pp. 2818–2826.
[17] K. He and et al., "Identity mappings in deep residual networks," in *ECCV*. Springer, 2016, pp. 630–645.
[18] M. Sandler and et al., "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018, pp. 4510–4520.
[19] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *NSDI*, 2017, pp. 613–627.
[20] N. Hasabnis, "Auto-tuning tensorflow threading model for cpu backend," in *MLHPC*. IEEE, 2018, pp. 14–25.
[21] Y. E. Wang, C.-J. Wu, X. Wang, K. Hazelwood, and D. Brooks, "Exploiting parallelism opportunities with deep learning frameworks," *TACO*, vol. 18, no. 1, pp. 1–23, 2020.
[22] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *ISLPED'17*, 2017, pp. 1–6.
[23] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *HPCA*, 2020, pp. 193–206.
[24] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis, "Rambo: Resource allocation for microservices using bayesian optimization," *IEEE CAL*, vol. 20, no. 1, pp. 46–49, 2021.