Fast and Efficient DNN Deployment via Deep Gaussian Transfer Learning

Qi Sun[1], Chen Bai[1], Tinghuan Chen[1], Hao Geng[1], Xinyun Zhang[2], Yang Bai[1], Bei Yu[1]
[1]The Chinese University of Hong Kong [2]SmartMore
{qsun,cbai,thchen,hgeng,ybai,byu}@cse.cuhk.edu.hk, xinyun.zhang@smartmore.com

Abstract

Deep neural networks (DNNs) have been widely used recently while their hardware deployment optimizations are very time-consuming and the historical deployment knowledge is not utilized efficiently. In this paper, to accelerate the optimization process and find better deployment configurations, we propose a novel transfer learning method based on deep Gaussian processes (DGPs). Firstly, a deep Gaussian process (DGP) model is built on the historical data to learn empirical knowledge. Secondly, to transfer knowledge to a new task, a tuning set is sampled for the new task under the guidance of the DGP model. Then DGP is tuned according to the tuning set via maximum-a-posteriori (MAP) estimation to accommodate for the new task and finally used to guide the deployments of the task. The experiments show that our method achieves the best inference latencies of convolutions while accelerating the optimization process significantly, compared with previous arts.

1. Introduction

Deep neural networks (DNNs) have shown great successes in various application scenarios [1, 2, 3, 4]. However, the enormous computational intensities and heavy data communications result in great challenges to inference. In recent years, great efforts have been made to accelerate the inference from various perspectives, including quantization [5, 6, 7], pruning [8, 9, 10, 11], optimization of deployment configurations [12, 13, 14, 15, 16], hardware-guided model training [17, 18], neural architecture search [19, 20], and *etc*. Many different platforms have been tested, *e.g.*, mobile devices [21, 22], FPGAs [23, 24], and GPUs [16, 25].

In this paper, we focus on the optimization of deployment configurations. Configurations represent the resource allocations, scheduling, binding of DNN models on hardware platforms, and *etc*. Traditionally, these optimization methods are tightly coupled with hardware architectures and model structures [26, 27, 28]. These methods usually propose some analytical formulations to model the latencies, and characterize the target DNN models and hardware platforms with respect to some properties, *e.g.*, the sizes of layers, and the capacities of buffers. Therefore, these methods cannot be flexibly adapted to different models. Further, some general deployment frameworks are developed, *e.g.*, Halide [12] and TVM [13], which use some auto-tuning algorithms to automatically find the optimal deployment configuration for any given model and hardware platform. For example, XGBoost [29] is used to build a boosted decision tree to predict the deployment performance of configurations. Simulated annealing (SA) is used as the solution searching algorithm. AutoTVM [14], which integrates the above algorithms, is an automatic optimization framework in TVM [13] and achieves outstanding performance. GGA [16] takes advantage of a guided genetic algorithm (GGA) to explore the candidate configurations, under the guidance of an artificially designed scoring calculator. CHAMELEON [15] proposes to use a proximal policy reinforcement learning algorithm to learn the actions to search the configuration space progressively.

However, these automatic frameworks are still unsatisfying. Firstly, the optimization process is slow, resulting from the large configuration space and the time-consuming compilation process. Usually, the configuration space contains more than millions of configurations, *e.g.*, more than 200 million in the first layer of VGG-16. It is inevitable to traverse lots of configurations to guarantee the performance of the searching algorithm. It also takes a long time to compile a configuration and do the inference to get the real on-board latency. Therefore, the overall optimization process is very slow, *e.g.*, longer than a whole day. Secondly, although lots of efforts are required to optimize the deployments of various DNN models, explicit empirics have seldom been drawn from the historical data. Despite that many duplicated works have been done to deploy some models, we usually start from scratch to optimize new models even though they are very similar to what has been deployed before. It is believed that with prior empirics, we can further improve the inference performance. CHAMELEON [15] leverages reinforcement learning to learn the evolution rules of the deployment configurations from the historical data. However, the experimental results show that the performance improvements mainly rely on adaptive sampling (AS) which adjusts the searching scope adaptively, instead of the policies of reinforcement learning. The guided genetic algorithm (GGA) [16] explores the candidate configurations to evaluate the similarities between the new layers and the history data, so as to guide the genetic evolution process. However, this method relies on complex and tricky evolution rules and the high-quality scoring calculator of the similarities. These disadvantages make it hard to be popularized in practical scenarios. And its deployment configurations have worse inference latencies compared with CHAMELEON [15]. Meanwhile, engineers are looking forward to the advent of automatic optimization flows without human interference. Ideally, the inputs of an automatic optimization flow are the historical tuning data with no need for manually designed rules.

To counteract these problems, on the one hand, it is urgent to find an accurate method to estimate the performance quickly without interacting with hardware to compile the model and do the inference. On the other hand, historical information should be fully utilized to guide the deployments of new models. In this paper, we propose a novel automatic optimization framework based on deep Gaussian transfer learning. Firstly, a deep Gaussian process (DGP) model is built on the historical optimization data to learn the hidden knowledge related to model structures, hardware characteristics, optimal deployment strategies, and *etc*. Stochastic variational inference is adopted to optimize the DGP. Secondly, when deploying a new DNN model, some efficient initial configurations of this new model are sampled under the guidance of the prior knowledge in the pre-trained DGP model. Maximum-a-posteriori (MAP) estimation is applied to tune the DGP model according to these initial configurations, to make the DGP model accommodate for the new task with no loss of the hidden knowledge. Finally, the tuned DGP model is used as a replacement to the time-consuming compilations and on-board inferences during optimization, to predict the performance values of new configurations accurately. Our tuned DGP model accelerates the optimization process remarkably while reducing the inference latency of the final model deployment simultaneously. We test our method on various types of convolutional layers and networks and results show that our method outperforms the state-of-the-art baselines significantly.

The remainder of this paper is organized as the following. Section 2 recaps the preliminaries. Section 3 illustrates our motivations and deep Gaussian transfer learning algorithm. Section 4 demonstrates the experiments and results. Finally, we conclude this paper in Section 5.

2. Preliminaries

DNN layers can be represented as several for-loops. Typically, convolutional operations can be represented as a seven-level for-loop, as shown in Figure 1. It is important to organize the hardware resources to conduct communications and computations and schedule these loops, *i.e.*, to determine an optimal deployment configuration.

```
for b in range(0, B):
  for o in range(0, M):
    for i in range(0, N):
      for h in range(0, H):
        for w in range(0, W):
          for kh in range(0, KH):
            for kw in range(0, KW):
              Out[b][o][h][w] += W[o][i][kh][kw]
                    × In[b][i][h+kh][w+kw]
```

Figure 1: A seven-level for-loop of a direct convolutional operation. B: batch size, M: number of output channels, N: number of input channels, H: height of features, W: width of features, KH: height of kernels, KW: width of kernels. The size of weight tensor is $[M, N, KH, KW]$ and the size of input tensor is $[B, N, H, W]$.

2.1. Deployment of DNN models

The NVIDIA CUDA [30] is taken as an example to explain the programming abstraction architecture on GPU, as shown in Figure 2. The programming architecture is composed of grids, blocks, and threads, and some memories. It provides fine-grained data parallelism, thread parallelism, nested within coarse-grained data parallelism, and task parallelism. The dense computational task is partitioned into smaller sub-tasks that can be conducted independently in parallel in these blocks. Following the single instruction multiple threads (SIMT) mechanism, each block is partitioned into a group of threads that can run the same code on different data synchronously. Except for GPU, other platforms including FPGA and ASIC have similar processing engines to conduct the DNN computations. Figure 3 is taken as an example to illustrate how to partition the workloads of DNN models and map them to hardware. For simplicity, the input tensor and weight tensor are represented as matrices with sizes $N \times B$ and $M \times N$, respectively. Firstly, the input and weight are split into small rectangles, with sizes *step* $\times$ *block-factor* and *block-factor* $\times$ *step*. The size of the corresponding outputs is *block-factor* $\times$ *block-factor*. To get the result of each output rectangle, its corresponding input and weight rectangles are assigned to a CUDA block to conduct the computations. Secondly, the computations are further split into *block-factor* $\times$ *block-factor* threads. Then these threads are assigned into some virtual groups to be scheduled by the CUDA runtime system.

For clearness, all of the deployment settings (*e.g.*, bindings of blocks, and threads) to be determined are encoded as the attributes of a feature vector which is termed as *deployment configuration*. A deployment configuration can be

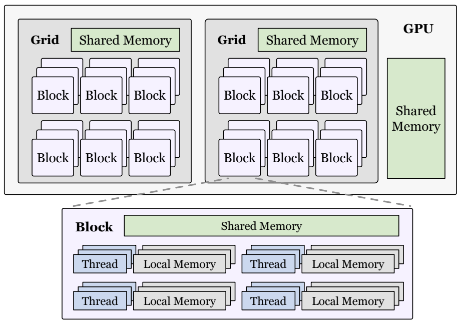


Figure 2: A brief CUDA programming architecture [30], composed of grids, blocks, threads, and some memories.

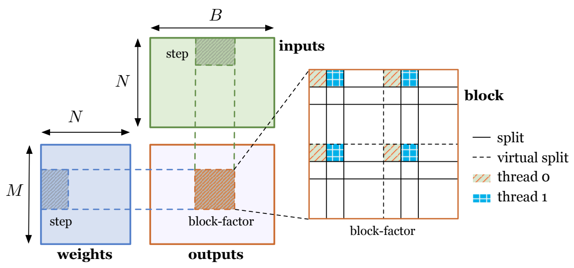


Figure 3: The computation workloads are partitioned into blocks and then further split to threads and virtual threads.

denoted as a feature vector $\boldsymbol{x}$.

2.2. General Automatic Deployment Flow

To determine the optimal deployment configuration, some automatic flows are developed, among which TVM [13] is widely used. In TVM, deployment configurations of layers in a DNN model are optimized layer by layer. Each of the for-loops on M, H, and W is split into four sub-loops. These four sub-loops are mapped to blocks, virtual threads, threads, and in-thread-for-loops, respectively. The bound of each sub-loop reflects the number of the allocated hardware resources. Each of the for-loops on N, KH, and KW is split into two sub-loops. These two sub-loops are mapped to threads, and in-thread-for-loops, respectively. The detailed information of deployment configurations is in the appendix. To determine the number of resources allocated to these sub-loops, *i.e.*, the bounds of these sub-loops, a comprehensive search space is defined, in which all of the possible configurations to the resource allocations are contained. The search space is usually composed of millions of configurations.

3. Transfer Learning Based on Deep Gaussian Processes

To avoid confusion, in the following, the *model* refers to our proposed DGP model, and the DNN model to be deployed on hardware is named as a *task*.

3.1. Motivations

In our problem, there are some great challenges, including the undersized available dataset resulting from the time-consuming design flow, and the uncertainties with respect to the characteristics of hardware and models which are hard to measure. Deep learning methods achieve outstanding results on many regression problems, but they are prone to be overfitted with a lack of large training dataset in our problem and be overconfident. Previous researches have shown that as the width of a one-hidden-layer neural network increases to infinity, the network converges to a Gaussian process (GP) model [31, 32, 33], which is a powerful nonparametric distribution and has wide applications [34, 35, 36]. GP method grows in complexity to suit the data and is robust enough to the overfitting on small datasets while providing reasonable predictions as well as uncertainty estimations. However, the GP models are limited by the expressiveness of kernel functions. Learning on a large and richly parameterized space of kernels is expensive, and approximations are at risk of overfitting [37, 38]. A deep Gaussian process (DGP) is a hierarchical composition of GPs that can overcome the limitations of GPs while retaining the advantages [39]. It can be regarded as a multi-layer neural network with multiple, infinitely wide hidden layers [40]. The mapping between layers is parameterized by a GP, and consequently, DGP can provide powerful uncertainty estimations. It performs input warping or dimensionality compression or expansion and automatically learns to construct a kernel that works well on the data. With these advantages, the DGP model is adopted in this paper, as the performance estimator in the optimization process of deployment configurations.

Considering the diversities and relationships between deployment tasks, it is imperative to transfer the knowledge learned in the source domain (historical task) to the target domain (new task). Some kernel learning methods are used as transfer learning approaches to learn scalable, expressive, and flexible kernels [41, 42, 43]. These methods rely on retraining or joint learning on a large number of tuning points of the new tasks. Note that we want to accelerate the searching process, therefore the slow joint learning and data collections are infeasible in our situation. These transfer learning approaches are also highly coupled with their regression or classification methods, which hinders flexibility. Traditionally, Gaussian process models are fitted from scratch via maximum likelihood estimation. In this paper, we propose a novel transfer learning algorithm based on the maximum-a-posteriori (MAP) estimation. The knowledge learned on history is used as the prior of DGP. Then DGP is tuned via MAP. This has similar philosophies with [44, 45], which also introduce a prior in the model and then calibrate it via posterior. Thanks to the knowledge learned on history, our method does not require much data on the new task. MAP is also easy to be solved with low workloads, so as to accelerate the search of optimums and improve the quality simultaneously. With these advantages, MAP has been widely used recently, *e.g.*, reinforcement learning [46], structured prediction [47], and statistical inference [48].

3.2. Our Automatic Optimization Framework

The overall optimization framework is shown in Figure 4. The DNN tasks are optimized layer by layer, following previous arts [13, 15, 16]. Before starting to optimize a new task, a deep Gaussian process model is built on the historical optimization data. The DGP preparation step is task-independent, *i.e.*, the pre-trained DGP model can be used to deploy other tasks. In Figure 4, the DNN task is represented as a graph, in which each node is a layer. For each layer, a searching space $\mathcal{D}$ containing all of the configurations of this layer is generated. In the tuning stage, the pre-trained DGP model is utilized as the criterion to sample some efficient initial configurations from $\mathcal{D}$. These initial configurations are then compiled and deployed to get their on-board performance values. These configurations and performance values are denoted as a tuning set. The hyper-parameters of the pre-trained DGP model are introduced as the prior. Maximum-a-posteriori (MAP) estimation is used to tune the pre-trained DGP model under the guidance of the tuning set. The tuned DGP model is then adopted as the performance estimator in the third stage (*i.e.*, the optimum searching stage). Various algorithms can be applied here as the searching algorithm to find optimal configurations. In experiments, we use simulated annealing as the searching algorithm. The previous arts [13, 15, 16] interact with hardware iteratively in the searching process to obtain the real on-board performance. By contrast, our tuned DGP can take the place of the real hardware and report the predicted performance, so as to accelerate the searching remarkably. All of the configurations found by the searching algorithm are recorded and the final optimal deployment configuration for this layer is selected from the record. The pseudo-code of our framework is provided in the appendix.

3.3. Deep Gaussian Processes with Stochastic Variational Inference

Denote our task as $f : \boldsymbol{x} \rightarrow y$, with the deployment configuration vector $\boldsymbol{x}$ and its performance value y. The historical optimization record is $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{y}\}$, with $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and $\boldsymbol{y} = \{y_1, \ldots, y_N\}$. For a single layer Gaussian process, the non-parametric Gaussian process places a GP prior over the value function f as $f(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}'))$, where $\mu(\cdot)$ is the mean function and $k(\boldsymbol{x}, \boldsymbol{x}')$ is the kernel function. $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})$ denotes the kernel matrix, *i.e.*, $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Given the

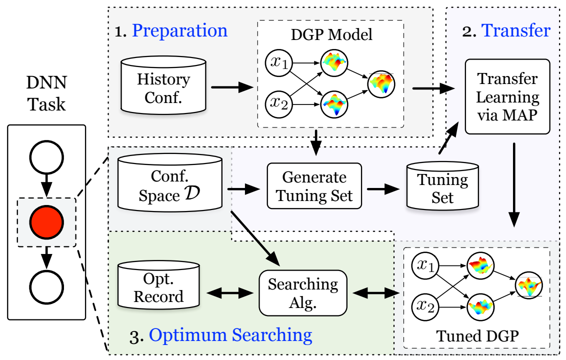


Figure 4: Our automatic optimization framework, consists of three stages, *i.e.*, stage 1: DGP model preparation based on the history data, stage 2: transfer knowledge to new DNN layers, and stage 3: optimal configuration searching.

historical data $\mathcal{D}$, $\boldsymbol{y}$ is assumed to be influenced by the zero-mean Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma_e^2)$, *i.e.*, $y_i = f(\boldsymbol{x}_i) + \epsilon$. The noise is indispensable to characterize the hardware uncertainties which may be caused by real-time workloads, temperature fluctuation, and *etc*. The function values with respect to $\boldsymbol{X}$ are denoted as a vector $\boldsymbol{f}$. Denote the hyper-parameters as $\boldsymbol{\theta}$, including noises and parameters in the kernel function. The marginal likelihood takes the form shown in Equation (1).

$$\mathcal{P}(\boldsymbol{y}|\boldsymbol{\theta}) = \prod_{i=1}^{N} \int p(y_i|f_i)p(f_i)\mathrm{d}f_i = \mathcal{N}(\boldsymbol{\mu}, \tilde{\boldsymbol{K}}), \tag{1}$$

where $\boldsymbol{\mu}$ is the mean vector, $\tilde{\boldsymbol{K}} = \boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X}) + \sigma_e^2 \boldsymbol{I}_N$, and $\boldsymbol{I}_N$ is the identity matrix.

A DGP model stacks multiple single-layer Gaussian processes. The outputs of the previous GP layer are the inputs of the next GP layer. For a DGP model comprised of L layers, denote the value functions of these L layers as $\{f^1, \cdots, f^L\}$. Correspondingly, the function values on inputs $\boldsymbol{X}$ are $\{\boldsymbol{f}^1, \cdots, \boldsymbol{f}^L\}$. Here $\boldsymbol{f}^0$ is defined as $\boldsymbol{X}$. The hyper-parameters in the l-th layer are represented as $\boldsymbol{\theta}^l$. Based on the definitions of the single-layer Gaussian process, the prior of a DGP model comprising L layers can be written as Equation (2).

$$\begin{aligned}&\mathcal{P}(\boldsymbol{f}^l) = \mathcal{N}(\boldsymbol{\mu}^l, \boldsymbol{K}^l),\ l = 1, \cdots, L,\\ &\mathcal{P}(\boldsymbol{y}, \{\boldsymbol{f}^1, \cdots, \boldsymbol{f}^L\}) = \prod_{i=1}^{N} \mathcal{P}(y_i|\boldsymbol{f}_i^L; \boldsymbol{\theta}^L) \prod_{l=1}^{L} \mathcal{P}(\boldsymbol{f}^l|\boldsymbol{f}^{l-1}; \boldsymbol{\theta}^l),\end{aligned} \tag{2}$$

where the hyper-parameters $\boldsymbol{\theta}^l$ are solved via maximum likelihood estimation which is computationally expensive. For the training set with N configurations, the computation complexity of the gradients of $\tilde{\boldsymbol{K}}$ with respect to $\boldsymbol{\theta}^l$ is $\mathcal{O}(N^3)$. Besides, the marginal density is unavailable in closed form or requires exponential time to compute [49], thus making the inference hard.

Inspired by recent works on posterior approximations of sparse Gaussian approximations, the stochastic variational inference [49] is employed to accelerate the computations of DGPs in this paper. The key technique is to introduce an inducing configuration set $\boldsymbol{Z} = \{\boldsymbol{z}^1, \cdots, \boldsymbol{z}^L\}$ with $|\boldsymbol{z}^l| \ll N$ and $\boldsymbol{z}^1 \subset \boldsymbol{X}$. Denote the function values at configurations $\boldsymbol{z}^l$ as $\boldsymbol{u}^l$ in the l-th layer. The basic assumption is that $\{\boldsymbol{u}^l\}_{l=1}^L$ is a sufficient statistic for $\{\boldsymbol{f}^l\}_{l=1}^L$, so that the real posterior $\mathcal{P}(\{\boldsymbol{f}^l, \boldsymbol{u}^l; \boldsymbol{\theta}^l\}_{l=1}^L|\boldsymbol{y})$ can be approximated given a Gaussian distribution $\mathcal{Q}(\{\boldsymbol{u}^l\}_{l=1}^L)$. To achieve the best $\{\boldsymbol{u}^l\}_{l=1}^L$, KL divergence between $\mathcal{P}(\{\boldsymbol{f}^l, \boldsymbol{u}^l; \boldsymbol{\theta}^l\}_{l=1}^L|\boldsymbol{y})$ and $\mathcal{Q}(\{\boldsymbol{f}^l, \boldsymbol{u}^l\}_{l=1}^L)$ is minimized with respect to the selection of $\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L$, as shown in Formulation (3).

$$\min_{\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L} \mathcal{KL}\left(\mathcal{Q}(\{\boldsymbol{f}^l, \boldsymbol{u}^l\}_{l=1}^L) \| \mathcal{P}(\{\boldsymbol{f}^l, \boldsymbol{u}^l; \boldsymbol{\theta}^l\}_{l=1}^L|\boldsymbol{y})\right). \tag{3}$$

Formulation (3) can be transferred to be the equivalent formulation as follows:

$$\begin{aligned} &\max_{\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L} \log \mathcal{P}(\boldsymbol{y}|\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L) \\ = &\max_{\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L} \left[\sum_{i=1}^N \mathbb{E}_{\mathcal{Q}(f_i^L|\boldsymbol{u}; \boldsymbol{x}_i, \boldsymbol{Z})}[\log \mathcal{P}(y_i|f_i^L; \boldsymbol{\theta}^L)]\right] \\ &- \sum_{l=1}^L \mathcal{KL}[\mathcal{Q}(\boldsymbol{u}^l) \| \mathcal{P}(\boldsymbol{u}^l; \boldsymbol{\theta}^l)], \end{aligned} \tag{4}$$

where $\mathcal{P}(\boldsymbol{y}|\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L)$ is the likelihood function. The model hyper-parameters $\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L$ are solved via Formulation (4). For simplicity, denote $\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}$ as $\tilde{\boldsymbol{\theta}}^l$. Until now, we have finished the training of our DGP model based on the historical data $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{y}\}$.

In this paper, the DGP model together with stochastic variational inference grows in complexity to suit the historical data and is robust enough to provide reasonable errors that would result from hardware or system uncertainties [40, 39, 49]. It also has a greater capacity to generalize and contains more hidden information compared with previous arts. The experimental results show us an outstanding performance by using our DGP and its high transferability.

3.4. Transfer Knowledge to New Tasks

To transfer the hidden knowledge and empirics from the known historical tasks (*a.k.a.*, source tasks) to new tasks (*a.k.a.*, target tasks), two steps are required, *i.e.*, finding a good initial tuning data set and choosing a fast and efficient transfer learning algorithm.

Firstly, to guarantee that adequate knowledge is learned for the new task, it is crucial to find a good tuning data set. Randomly picking some initial configurations from the extremely large design space would introduce some illegal configurations (*i.e.*, with performance values equal to zero) which cannot help us but wastes lots of time to compile them. Besides, there is no guarantee that the historical data set is large enough to cover the data distribution of the target task. The target task would also possibly have a higher upper bound of performance values, which means the mean value of the historical data might be smaller than the mean value of the target task. Therefore, the tuning data set should contain configurations with performance values as higher as possible, to calibrate the mean function.

To handle these, the DGP model learned from the historical data is used as the empirical criterion to select suitable initial configurations for the new task. A set which is large enough is randomly sampled from the search space and then fed into the DGP model to get the predicted performance. We sort these initial configurations according to their predicted performance values and the configurations with top s performance values are chosen as the tuning points, *i.e.*, $\boldsymbol{X}^t = \{\boldsymbol{x}_1^t, \cdots, \boldsymbol{x}_s^t\}$. The configurations in $\boldsymbol{X}^t$ are compiled and deployed on real hardware to get the real performance set $\boldsymbol{y}^t = \{y_1^t, \cdots, y_s^t\}$. Denote $\{\boldsymbol{X}^t, \boldsymbol{y}^t\}$ as $\mathcal{D}^t$, and then $\mathcal{D}^t$ is used as the tuning set to calibrate the empirical DGP model. Intuitively, a tuning set with high diversities is better. However, in our context, the configuration space is too large and only small parts have good performance. Our target is to find the optimal configurations instead of characterizing the whole configuration space. In other words, we are interested in a small part of the solution space with high performance. Besides, compared with the large space, the size of the randomly sampled set and the number of the sorted configurations are small, a basic situation is that these sampled configurations will always scatter with high diversities in the space. Therefore, finding a better initial tuning set via our DGP is wise with no harm to the diversities.

Secondly, a fast and efficient transfer learning algorithm based on MAP is proposed to tune the DGP model with $\mathcal{D}^t$. As mentioned above, the widely-used transfer learning algorithms [41, 42, 43] are unsuitable in our situations for several reasons. For the target task, with the help of MAP, the model parameters are optimally determined by combining the hidden knowledge (in the form of the parameters $\tilde{\boldsymbol{\theta}}_l$) and $\mathcal{D}^t$. For the convenience of explanation, we omit the layer indices to lighten the notations. Denote all of the parameters in the source task DGP model as $\tilde{\boldsymbol{\theta}}$ and the parameters for target task as $\hat{\boldsymbol{\theta}}$. According to the Bayes' theorem, MAP is to find the optimal value of $\hat{\boldsymbol{\theta}}$ (*i.e.*, most likely to occur) to maximize the posterior distribution $\mathcal{P}(\hat{\boldsymbol{\theta}}|\boldsymbol{y}^t)$. Specifically, $\mathcal{P}(\hat{\boldsymbol{\theta}}|\boldsymbol{y}^t)$ follows Formulation (5).

$$\mathcal{P}(\hat{\boldsymbol{\theta}}|\boldsymbol{y}^t) \propto \mathcal{P}(\hat{\boldsymbol{\theta}}) \cdot \mathcal{P}(\boldsymbol{y}^t|\hat{\boldsymbol{\theta}}), \tag{5}$$

where $\mathcal{P}(\boldsymbol{y}^t|\hat{\boldsymbol{\theta}})$ is the likelihood function in Formulation (4). The prior of $\hat{\boldsymbol{\theta}}$ is assumed to follow a Gaussian distribution with $\tilde{\boldsymbol{\theta}}$ as the mean value [45]. To accelerate the computation, the MAP is implemented as an L2 regularization term of $\hat{\boldsymbol{\theta}}$ and $\tilde{\boldsymbol{\theta}}$, *i.e.*, $\|\hat{\boldsymbol{\theta}} - \tilde{\boldsymbol{\theta}}\|_2^2$. The objective function to tune the parameters is defined as Formulation (6).

$$\max_{\hat{\boldsymbol{\theta}}} \quad \log \mathcal{P}(\boldsymbol{y}^t | \hat{\boldsymbol{\theta}}) - \lambda \|\hat{\boldsymbol{\theta}} - \tilde{\boldsymbol{\theta}}\|_2^2, \tag{6}$$

where λ is a hyper-parameter. Theoretically, L2 regularization is equivalent to MAP inference with a Gaussian prior on the parameters [50]. Compared with the traditional GP methods which are fitted from scratch, our method with prior $\tilde{\boldsymbol{\theta}}$ does not require too much data and saves time.

An important characteristic of deployment is that various computation operations would have a non-negligible influence on the communication modes, resource allocations, and *etc*. For example, depthwise convolutions and direct convolutions have distinct computation patterns. These characteristics are hard to be summarized as a unified rule even for senior engineers. To guarantee the performance of our flow, DNN layers are categorized into some groups according to their types. The knowledge is learned and transferred within each group.

4. Experiments

We implement our flow based on GPyTorch [51] and embed it into TVM to validate the performance. Some ablation studies are conducted. Layer-wise and model-wise performance are analyzed and compared with the state-of-the-art. Due to space limitations, we present some important settings and representative results in the paper, and more details and results can be referred to the appendix.

4.1. Experimental Settings

Our experiments are running on an Intel(R) Xeon(R) E5-2680 v4 CPU@ 2.40GHz. The hardware platform is an NVIDIA GeForce GTX 1080Ti GPU and the CUDA version is 9.0.176. For fair comparisons, models tested in previous work [14, 15, 16] are tested, including AlexNet [52], ResNet-18 [53], and VGG-16 [54]. Further, MobileNet-v1 [55] is tested. Note that the current deployment flows [14, 15, 16] optimize the DNN models layer by layer. The optimization algorithms and processes are independent of the model structure, therefore simple model structures are enough to validate our method with no need of using more complicated DNN models. The representative DNN layers widely used in both industries and academia are covered in these models, including convolutional layers, residual blocks, depthwise separable convolutional layers, and *etc*. For the layers with the same structures, only the first of them will be optimized and others directly use the same configuration for this layer. Besides, same with [14, 15, 16], we focus on the optimizations of various convolutional layers, and other types of layers are skipped, *e.g.*, fully connected layers and pooling layers. These other layers directly use the settings provided by TVM.

AutoTVM [14], integrating XGBoost, simulated annealing, and *etc.*, is used as the baseline. Besides, two outstanding baselines are also compared, including DAC'20 GGA [16] which uses a well-designed heuristic guided genetic algorithm, and ICLR'20 CHAMELEON [15] which is based on reinforcement learning and adaptive sampling algorithm. In our method, we use the simulated annealing in TVM as the searching algorithm and follow the same settings as AutoTVM and CHAMELEON. Our DGP is used as the performance estimator and a replacement to GPU during configuration searching, as mentioned in Section 3.2. The radial basis function is adopted as the kernel function. The number of inducing points in variational inference is 128. Notably, the experimental platforms and software versions are distinct compared with other works, which would have a significant effect on the search time and the performance of the final deployments. For fairness, the results are expressed as ratios to the results of AutoTVM.

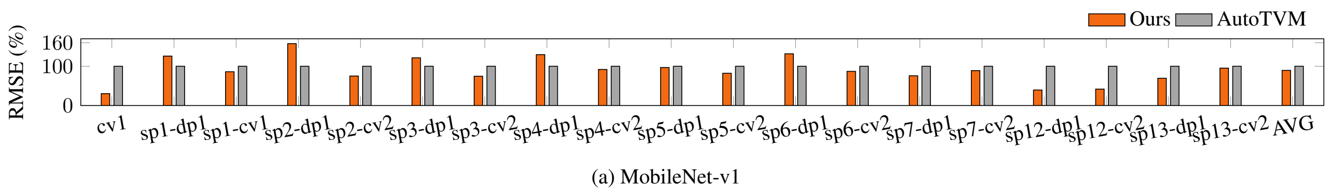
To illustrate the performance, three metrics are used, *i.e.*, Giga floating operations per second (GFLOPS), the reduction of the inference latency, and the reduction of the search time to find the optimal configuration. GFLOPS measures the number of floating-point operations conducted by the hardware per second during executing the model. It is used as the optimization objective for each layer in our method and the baselines. Inference latency is the final end-to-end on-board inference time of the whole model. Search time is the overall time cost to optimize the deployment of the whole DNN model, including the interactions with hardware and model tuning.

4.2. Layer Groups and Historical Data

As mentioned above, to guarantee the transferability of prior knowledge, the DNN layers are categorized into some groups. In this paper, we choose three criteria, including layer type (*e.g.*, direct convolution, or depthwise separable convolution), kernel size (*e.g.*, 3×3, or 7×7), and padding type (*e.g.*, no padding, or padding size = 1). These criteria are fundamental factors that would have a great influence on the deployment performance and usually bother engineers. According to these criteria, VGG-16 has 1 group, while ResNet-18, AlexNet, and MobileNet-v1 have 4, 3, and 4 groups, respectively. The first layer of each group is deployed via AutoTVM and the configurations explored by AutoTVM in this process are collected as the historical data for this group. This makes our method more practical in demanding application scenarios.

4.3. Ablation Studies on the Proposed DGP

We perform ablation studies on our pre-trained DGP model, *i.e.*, evaluate the results of stage 1 in Figure 4. The accuracies of directly using DGP trained on the historical data to predict the performance of configurations of new



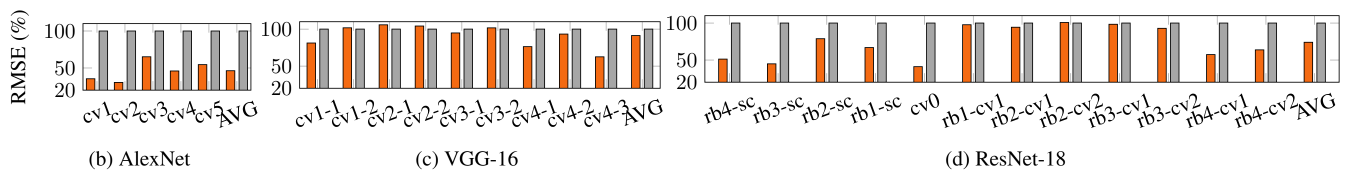


Figure 5: RMSE of our predicted GFLOPS, the data are expressed as the ratios to the results of XGBoost in AutoTVM. Here, our DGP is directly used to predict the GFLOPS of new tasks without tuning. cv: convolution, rb: residual block, sc: shortcut, sp: separable convolution, dp: depthwise convolution.

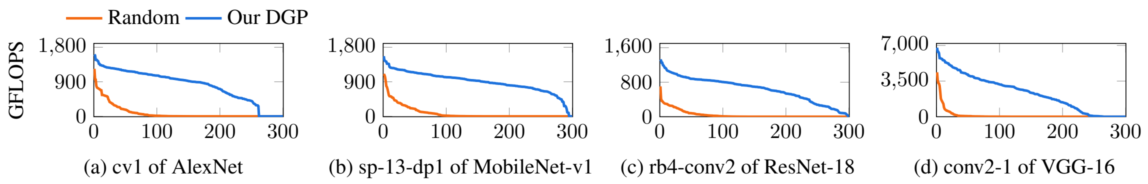


Figure 6: The randomly sampled tuning set and the set selected according to DGP. The data are in descending order. There are 300 configurations in each tuning set, and the X-axis is the index of the configuration.

layers are plotted, in comparison with the prediction performance of the XGBoost used by AutoTVM. For fair comparisons, these two methods use the same training data, as mentioned in Section 4.2. After training, they are directly used to predict the performance without tuning. The root-mean-square error (RMSE) of the predicted GFLOPS values is to characterize the prediction error. The results are shown in Figure 5. The prediction accuracy of our DGP on direct convolutional layers outperforms XGBoost remarkably, no matter whether with padding or not, or with various sizes of kernels, or different sizes of strides. Our DGP wins on most of the depthwise convolutional layers. As to the performance of residual blocks, our method is also the superior one. On these four models, our average results are the best. It is demonstrated that our DGP models are able to learn enough prior knowledge of the hidden characteristics of the hardware architecture, model structures, and *etc*.

As mentioned above, the pre-trained DGP is used as the empirical criterion to select a suitable initial configuration set for the subsequent tuning stage. Note that our target is to learn the good configurations instead of the whole configuration space. Using our DGP will help choose the useful configurations and will teach the model to learn more about the characteristic of this layer. Examples of the sampled configurations and their on-board GFLOPS values are plotted in Figure 6. In experiments, the tuning set contains 300 configurations. Most of the configurations sampled via our DGP model are feasible and have continuous GFLOPS values. In comparison, most of the randomly sampled configurations are infeasible on hardware. Besides, the maximum GFLOPS of the random method is lower than ours which means the DGP tuned on the random set is unable to give higher prediction values for good configurations.

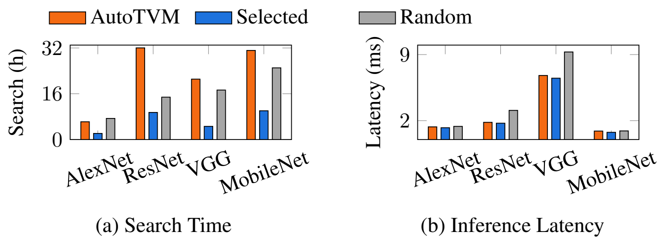


Figure 7: Comparisons between AutoTVM and ours. “Selected” means the tuning configurations are selected by using our pre-trained DGP as the criterion. “Random” means the tuning configurations are randomly sampled from the configuration space without any prior knowledge.

Table 1: Comparisons of Search Time and End-to-end Model Inference Latency

Model	AutoTVM [14]		ICLR'20 [15]			Ours				
	Search (h)	Inference (ms)	Search Redu. (%)	Inference Redu. (%)	HV	Search (h)	Search Redu. (%)	Inference (ms)	Inference Redu. (%)	HV
MobileNet-v1	31.14	0.8980	-	-	-	10.06	67.69	0.7664	**14.65**	**9.9168**
AlexNet	6.28	1.3467	72.16	5.88	4.2409	2.14	65.96	1.2537	**6.91**	**4.5573**
VGG-16	19.92	6.7847	82.56	3.44	2.8418	4.61	76.83	6.4972	**4.24**	**3.2556**
ResNet-18	32.04	1.8248	76.67	4.16	3.1915	9.47	70.43	1.7305	**5.17**	**3.6423**

4.4. Ablation Studies on the Transfer Learning

To prove the effectiveness of our transfer learning method, we compare the results of using the randomly sampled tuning set with the results of using the tuning set selected by our DGP (as mentioned in Section 3.4). The results are shown in Figure 7. The randomly sampled tuning sets increase the inference latencies significantly because the performance of most of the sampled configurations is unsatisfying. Randomly sampling a small number of configurations cannot introduce enough knowledge about the optimal configurations, but confuses the pre-trained DGPs.

4.5. Performance of the Whole framework

Some results with respect to the reduction of search time of the whole optimization process and reduction of model inference latency are analyzed here, compared with the state-of-the-art baselines. The detailed results are listed in Table 1. The reported latency is the average of latencies from 1800 on-board inference trials and is believed to be accurate enough. Usually, there is a trade-off between search time and model inference latency. To improve the model inference performance, more configurations are sampled and analyzed in the searching process. Consequently, the search time increases, and vice versa. For fair comparisons between these two closely related and interacting metrics, we introduce the concept of hypervolume (HV) [56]. Hypervolume is commonly adopted to measure the solutions of multi-objective optimization problems. The reduction ratios of inference latency and search time are multiplied, to measure the overall performance, as shown in Equation (7).

$$\text{HV} = \text{Redu. of Latency} \times \text{Redu. of Search Time} \times 100. \tag{7}$$

Here the HV value is multiplied by 100 to adjust the order of magnitude. The solution with a higher HV value is the better one. The results prove the superior performance of our method. Compared with ICLR'20 CHAMELEON [15], though our reductions in search times are not optimal, the reductions of the inference latencies are much better. Our overall results are much better than CHAMELEON, with respect to the HV values. In GGA [16], the authors reduce the search time of ResNet-18 by 93.17% and reduce the inference latency by 3.26%. Although they have the fastest search speed, the inference latency is the worst. The HV value of their method is 3.037, which is also worse than [15] and ours. Accelerating the search speed too aggressively does not worth the loss of the quality of results. The precise search times and inference latencies of VGG-16, AlexNet, and MobileNet-v1 are not provided in GGA [16]. For supplementary, the GFLOPS values of VGG-16 are plotted in Figure 8. Compared with AutoTVM, our method wins on most layers and has a better average GFLOPS value.

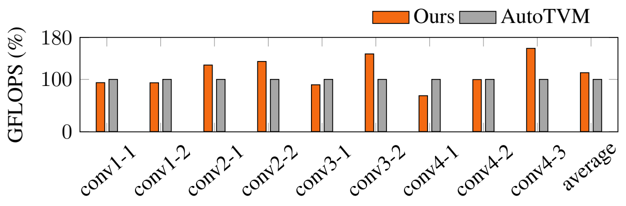


Figure 8: The ratios of the GFLOPS values of VGG-16.

Despite the existing trade-off between the searching time and the inference latency, on-chip inference latency is actually the most critical metric since the model deployment is "once for all", which means no matter how much time we spent to optimize the deployment, the faster on-chip inference is more important than the faster optimization process. From this perspective, our method also outperforms the baselines significantly.

5. Conclusion

In this paper, a transfer learning algorithm based on a deep Gaussian process (DGP) is proposed to optimize the deployment of DNN models, by using the historical information efficiently. The representative DNN layers and models are tested. Both the search time and inference latency are reduced simultaneously. The experiments show that our method outperforms the baselines remarkably.

Acknowledgment

This work is partially supported by SmartMore and ITF Partnership Research Programme (No. PRP/65/20FX).

References

[1] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. NIPS*, 2015, pp. 91–99. 1

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018. 1

[3] Y. Bai and W. Wang, "ACPNet: Anchor-Center Based Person Network for Human Pose Estimation and Instance Segmentation," in *Proc. ICME*, 2019, pp. 1072–1077. 1

[4] Q. Sun, A. A. Rao, X. Yao, B. Yu, and S. Hu, "Counteracting adversarial attacks in autonomous driving," in *Proc. ICCAD*, 2020, pp. 1–7. 1

[5] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "HAWQ: Hessian aware quantization of neural networks with mixed-precision," in *Proc. ICCV*, 2019, pp. 293–302. 1

[6] T. Ajanthan, P. K. Dokania, R. Hartley, and P. H. Torr, "Proximal mean-field for neural network quantization," in *Proc. ICCV*, 2019, pp. 4871–4880. 1

[7] J. Fang, A. Shafiee, H. Abdel-Aziz, D. Thorsley, G. Georgiadis, and J. H. Hassoun, "Post-training piecewise linear quantization for deep neural networks," in *Proc. ECCV*, 2020, pp. 69–86. 1

[8] Y. Zhou, Y. Zhang, Y. Wang, and Q. Tian, "Accelerate CNN via recursive Bayesian pruning," in *Proc. ICCV*, 2019, pp. 3306–3315. 1

[9] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun, "Metapruning: Meta learning for automatic neural network channel pruning," in *Proc. ICCV*, 2019, pp. 3296–3305. 1

[10] H. Tian, B. Liu, X.-T. Yuan, and Q. Liu, "Meta-learning with network pruning," in *Proc. ECCV*, 2020, pp. 675–700. 1

[11] T. Chen, B. Duan, Q. Sun, M. Zhang, G. Li, H. Geng, Q. Zhang, and B. Yu, "An efficient sharing grouped convolution via bayesian learning," in *IEEE TNNLS*, 2021, pp. 1–13. 1

[12] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in Halide," *ACM SIGGRAPH*, vol. 37, no. 4, pp. 139:1–139:13, 2018. 1

[13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI*, 2018, pp. 578–594. 1, 3, 4

[14] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Proc. NIPS*, 2018, pp. 3389–3400. 1, 6, 8

[15] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "CHAMELEON: Adaptive code optimization for expedited deep neural network compilation," in *Proc. ICLR*, 2020. 1, 2, 4, 6, 8

[16] J. Mu, M. Wang, L. Li, J. Yang, W. Lin, and W. Zhang, "A history-based auto-tuning framework for fast and high-performance DNN design on GPU," in *Proc. DAC*. IEEE, 2020, pp. 1–6. 1, 2, 4, 6, 8

[17] C. Li, T. Chen, H. You, Z. Wang, and Y. Lin, "HALO: Hardware-aware learning to optimize," in *Proc. ECCV*, 2020, pp. 500–518. 1

[18] E. Park and S. Yoo, "Profit: A novel training method for sub-4-bit MobileNet models," in *Proc. ECCV*, 2020, pp. 430–446. 1

[19] C. Gong, Z. Jiang, D. Wang, Y. Lin, Q. Liu, and D. Z. Pan, "Mixed precision neural architecture search for energy efficient deep learning," in *Proc. ICCAD*. IEEE, 2019, pp. 1–7. 1

[20] Z. Yuan, B. Wu, G. Sun, Z. Liang, S. Zhao, and W. Bi, "S2DNAS: Transforming static CNN model for dynamic inference via neural architecture search," in *Proc. ECCV*, 2020, pp. 175–192. 1

[21] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. ECCV*, 2018, pp. 784–800. 1

[22] X. Ma, W. Niu, T. Zhang, S. Liu, S. Lin, H. Li, W. Wen, X. Chen, J. Tang, K. Ma *et al.*, "An image enhancing pattern-based sparsity for real-time inference on mobile devices," in *Proc. ECCV*, 2020, pp. 629–645. 1

[23] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. CVPR*, 2019, pp. 8612–8620. 1

[24] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation," in *Proc. DAC*. IEEE, 2020, pp. 1–6. 1

[25] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and parallel GPU task scheduling for deep learning," *Proc. NIPS*, 2020. 1

[26] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. DAC*, 2017, pp. 29:1–29:6. 1

[27] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. ICCAD*, 2018, pp. 56:1–56:8. 1

[28] Q. Sun, T. Chen, J. Miao, and B. Yu, "Power-driven DNN dataflow optimization on FPGA," in *Proc. ICCAD*, 2019, pp. 1–7. 1

[29] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. KDD*, 2016, pp. 785–794. 1

[30] D. Kirk *et al.*, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104. 2, 3

[31] L. Jaehoon, B. Yasaman, N. Roman, S. S. Samuel, P. Jeffrey, and S. Jascha, "Deep neural networks as gaussian processes," in *Proc. ICLR*, 2018. 3

[32] A. G. d. G. Matthews, M. Rowland, J. Hron, R. E. Turner, and Z. Ghahramani, "Gaussian process behaviour in wide deep neural networks," in *Proc. ICLR*, 2018. 3

[33] M. E. E. Khan, A. Immer, E. Abedi, and M. Korzepa, "Approximate inference turns deep networks into gaussian processes," in *Proc. NIPS*, 2019. 3

[34] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016. 3

[35] W. Lyu, F. Yang, C. Yan, D. Zhou, and X. Zeng, "Batch Bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design," in *Proc. ICML*, 2018, pp. 3306–3314. 3

[36] C. Lo and P. Chow, "Multi-fidelity optimization for high-level synthesis directives," in *Proc. FPL*, 2018, pp. 272–2727. 3

[37] D. Duvenaud, J. Lloyd, R. Grosse, J. Tenenbaum, and G. Zoubin, "Structure discovery in nonparametric regression through compositional kernel search," in *Proc. ICML*, 2013, pp. 1166–1174. 3

[38] R. Calandra, J. Peters, C. E. Rasmussen, and M. P. Deisenroth, "Manifold Gaussian processes for regression," in *Proc. IJCNN*. IEEE, 2016, pp. 3338–3345. 3

[39] H. Salimbeni and M. Deisenroth, "Doubly stochastic variational inference for deep Gaussian processes," in *Proc. NIPS*, 2017, pp. 4588–4599. 3, 5

[40] T. Bui, D. Hernández-Lobato, J. Hernandez-Lobato, Y. Li, and R. Turner, "Deep Gaussian processes for regression using approximate expectation propagation," in *Proc. ICML*, 2016, pp. 1472–1481. 3, 5

[41] M. Kandemir, "Asymmetric transfer learning with deep Gaussian processes," in *Proc. ICML*, 2015, pp. 730–738. 3, 5

[42] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing, "Deep kernel learning," in *Proc. AISTATS*, 2016, pp. 370–378. 3, 5

[43] M. Patacchiola, J. Turner, E. J. Crowley, M. O'Boyle, and A. Storkey, "Deep kernel transfer in Gaussian processes for few-shot learning," *arXiv preprint arXiv:1910.05199*, 2019. 3, 5

[44] C. Louizos, K. Ullrich, and M. Welling, "Bayesian compression for deep learning," in *Proc. NIPS*, vol. 30, 2017. 3

[45] F. Wang, P. Cachecho, W. Zhang, S. Sun, X. Li, R. Kanj, and C. Gu, "Bayesian model fusion: large-scale performance modeling of analog and mixed-signal circuits by reusing early-stage data," *IEEE TCAD*, vol. 35, no. 8, pp. 1255–1268, 2016. 3, 5

[46] H. F. Song, A. Abdolmaleki, J. T. Springenberg, A. Clark, H. Soyer, J. W. Rae, S. Noury, A. Ahuja, S. Liu, D. Tirumala, N. Heess, D. Belov, M. Riedmiller, and M. M. Botvinick, "V-mpo: On-policy maximum a posteriori policy optimization for discrete and continuous control," in *Proc. ICLR*, 2020. 4

[47] A. Ghoshal and J. Honorio, "Learning maximum-a-posteriori perturbation models for structured prediction in polynomial time," in *Proc. ICML*. PMLR, 2018, pp. 1754–1762. 4

[48] T. Hazan, F. Orabona, A. D. Sarwate, S. Maji, and T. S. Jaakkola, "High dimensional inference with random maximum a-posteriori perturbations," *IEEE Transactions on Information Theory (TIT)*, vol. 65, no. 10, pp. 6539–6560, 2019. 4

[49] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal*

of the American Statistical Association, vol. 112, no. 518, pp. 859–877, 2017. 5

[50] I. Goodfelow, Y. Bengio, and A. Courville, "Deep learning (adaptive computation and machine learning series)," 2016. 6

[51] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, and A. G. Wilson, "GPyTorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration," in *Proc. NIPS*, 2018. 6

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012, pp. 1097–1105. 6

[53] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016, pp. 770–778. 6

[54] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *Proc. ICLR*, 2015. 6

[55] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017. 6

[56] L. While, P. Hingston, L. Barone, and S. Huband, "A faster algorithm for calculating Hypervolume," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 1, pp. 29–38, 2006. 8