Placeto: Efficient Progressive Device Placement Optimization

Ravichandra Addanki, **Shaileshh Bojja Venkatakrishnan**, **Shreyan Gupta**, **Hongzi Mao**, **Mohammad Alizadeh**

MIT Computer Science and Artificial Intelligence Laboratory
{addanki, bjjvnkt, shreyang, hongzi, alizadeh}@mit.edu

Abstract

We present Placeto, a reinforcement learning (RL) approach to efficiently find device placements for distributed neural network training. Unlike prior approaches that only find a device placement for a specific computational graph, Placeto can learn generalizable device placement policies that can be applied to *any* graph. We propose two key ideas in our approach: (1) we represent the policy as performing iterative placement improvements, rather than outputting a placement in one shot; (2) we use graph embeddings to capture the structural information of the computational graph, without relying on node labels for indexing. These ideas allow Placeto to train efficiently and generalize to unseen graphs. Our experiments show that Placeto can take up to $20\times$ fewer training steps to find placements that are on par with or better than the best placements found by prior approaches.

Introduction

The computational requirements for training neural networks has steadily increased in recent years. As a result, a growing number of applications [11, 15] use distributed training environments in which a neural network is split across multiple GPU and CPU devices. A key challenge for distributed training is how to split a large model across multiple heterogeneous devices to achieve the fastest possible training speed. Today device placement is typically left to human experts, but determining an optimal device placement can be very challenging, particularly as neural networks grow in complexity (e.g., many interconnected branches) or approach device memory limits. In shared clusters, the task is made even more challenging due to interference and variability caused by other applications.

Motivated by these challenges, a recent line of work [10, 9, 5] has proposed an automated approach to device placement based on reinforcement learning (RL). In this approach, a neural network policy is trained to optimize the device placement through repeated trials. For example, Mirhoseini et al. [10] use a recurrent neural network (RNN) to process a computational graph and predict a placement for each operation. They show that the RNN, trained to minimize computation time, produces device placements that outperform both human experts and graph partitioning heuristics such as Scotch [13]. Subsequent work [9] improved the scalability of this approach with a hierarchical model and explored more sophisticated policy optimization techniques [5].

Although RL-based device placement is promising, existing approaches have a key drawback: they require significant amount of training to find a good placement for each computational graph. For example, Mirhoseini et al. [10] report 12 to 27 hours of training time to find the best device placement for several vision and natural language models; more recently, the same authors report 12.5 GPU-hours of training to find a placement for the NMT model [9]. While this overhead may be acceptable in some scenarios (e.g., training a stable model on large amounts of data), it is undesirable in many cases. For example, high device placement overhead is problematic during model development, which can require many ad-hoc model explorations. Also, in a shared, non-stationary environment, it is important to make a placement decision quickly, before the underlying environment changes.

Existing methods have high overhead because they do not learn *generalizable* device placement policies. Instead they optimize the device placement for a *single* computational graph. Indeed, the training process in these methods can be thought of as a search for a good placement for one graph, rather than a search for a good placement *policy* for a class of computational graphs. Therefore, for a new computational graph, these methods must train the policy network from scratch. Nothing learned from previous graphs carries over to new graphs, neither to improve placement decisions nor to speed up the search for a good placement.

We present Placeto, a reinforcement learning (RL) approach to learn an efficient algorithm for device placement on any graph. Unlike prior work, Placeto is able to transfer a learned placement policy to unseen computational graphs, without extensive re-training. Placeto incorporates two key ideas to improve training efficiency and generalizability. First, it models the device placement task as finding a sequence of *iterative placement improvements*. Specifically, Placeto's policy network takes as input a current placement for a computational graph, and one of its node, and it outputs a device for that node. By applying this policy sequentially to all nodes, Placeto is able to iteratively optimize the placement. This placement improvement policy, operating on an explicitly-provided input placement, is simpler to learn than a policy representation that must output a final placement in one step.

Placeto's second idea is a neural network architecture that uses *graph embeddings* to encode the computational graph structure in the placement policy. Unlike prior RNN-based approaches, Placeto's neural network policy does not depend on the sequencing order of nodes or an arbitrary labeling of the graph (e.g., to represent adjacency information). Instead it naturally captures graph structure (e.g., parent-child relationships) via iterative message passing computations performed on the graph.

Our experiments empirically show that Placeto learns placement policies that are on par with or better than the RNN-based approach over three neural network models: Inception-V3 [17], NASNet [20], NMT [19]. However, it can learn these placement substantially faster than the RNN approach. Training Placeto from scratch, it finds near-optimal device placements with 2.6-4.2× fewer placements sampled than the RNN approach. Transferring a learned policy (e.g., trained for Inception) to unseen graphs provides further speedups, requiring 6-21× fewer placements sampled than prior approaches.

Design

Consider the computational graph $G(V, E)$ of a neural network, consisting of atomic computational operations (also referred to "ops") V, and data communication edges E. Each op $v \in V$ performs a specific computational function (e.g., convolution) on input tensors that it receives from its parent ops. For a set of devices $D = \{d_1, \ldots, d_m\}$, a *placement* for G is a mapping $\pi : V \rightarrow D$ that assigns a device to each op. The goal of device placement is to find a placement π that minimizes $\rho(G, \pi)$, the duration of G's execution when its ops are placed according to π. To reduce the number of placement actions, we partition ops into predetermined *groups* and place ops from the same group on the same device, similar to Mirhoseini et.al. [9]. Henceforth $G(V, E)$ refers to the graph of op groups.

MDP formulation. We model the placement problem as a Markov decision process (MDP) over a state space of all possible device placement configurations as follows. The initial state s_0 of the MDP consists of G with an arbitrary device placement for each op group $v \in V$. The action in step t outputs a new placement for the t-th node in G, based on s_{t-1}. The episode ends in $|V|$ steps when all nodes have been visited exactly once. We consider two approaches for assigning rewards: (1) assign a zero reward at each intermediate RL step and a final reward equal to the negative run time of the final placement; (2) assign intermediate rewards $r_t = \rho(s_{t+1}) - \rho(s_t)$ at the t-th RL round for each t, where $\rho(s)$ is the execution time of placement s. Intermediate rewards can help improve credit assignment in long training episodes and reduce variance of the policy gradient estimates [2, 12, 16]. However, training with intermediate rewards is more expensive, as it must determine the computation time for a placement at each step as opposed to once per episode.

Graph embedding. To compute placement actions, we need to encode the graph structured states as real-valued vectors. Inspired by recent works on inductive graph representation learning [3, 4, 7, 8], we present a graph embedding approach that processes the raw features associated with each node in three steps (Figure 1) —

1. *Computing per-group attributes* (Figure 1a). As raw features for each op group, we use the total execution time of ops in the group, total tensor size in the outputs, a one-hot encoding of the

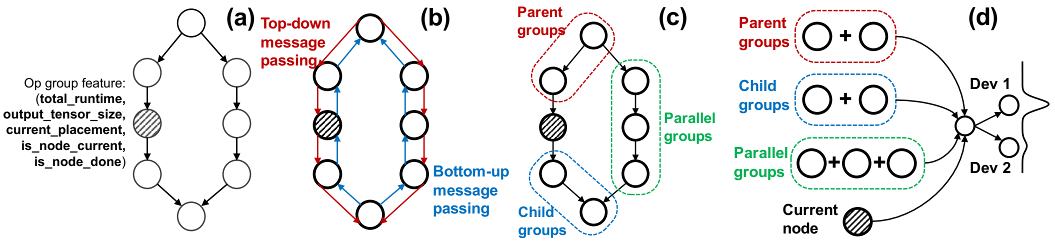

Figure 1: Placeto's graph embedding approach. It maps raw features associated with each op group to the device placement action. (a) Example computation graph of op groups. The shaded node is taking the current placement action. (b) Two-way message passing scheme applied to all nodes in the graph. (c) Partitioning the message passed (denoted as bold) op groups. (d) Taking a placement action on two candidate devices for the current op group.

placement for the group, a binary encoding of whether the current placement action is for this group, and a binary encoding of whether a placement action has already been made for the group. We collect the runtime of each op on each device from on-device measurements (Appendix 5).

2. *Local neighborhood summarization* (Figure 1b). Using the raw features on each node, we perform a sequence of two-way message passing iterations [4, 7] to aggregate neighborhood information for each node. Letting $\mathbf{x}_v$ denote the features of op group v, the message passing updates take the form $\mathbf{x}_v \leftarrow g(\sum_{u \in \xi(v)} f(\mathbf{x}_u))$, where $\xi(v)$ is the set of neighbors of v, and f, g are multilayer perceptrons with trainable parameters. We construct two directions (top-down from root groups and bottom-up from leaf groups) of message passings with separate parameters. The two directions aggregate different information affiliated with the graph structure (e.g., top-down messages can summarize the locality information of placed devices; bottom-up messages can summarize how much work in total exists in the child subtree). The parameters in the transformation functions f, g remain the same for message passing in each direction among all nodes and for all message passing steps. We repeat the message passing updates for ten times to propagate information. As shown in our experiments (§3), reusing the same message passing function everywhere provides a natural way to transfer the learned policy to unseen models.
3. *Pooling summaries* (Figures 1c and 1d). After message passing, we aggregate the node embeddings computed via message passing to create a global summary of the entire graph. Specifically, for the node v to take the current placement action, we perform three separate aggregations: on the set $S_{\text{parents}}(v)$ of nodes that can reach v, set $S_{\text{children}}(v)$ of nodes that are reachable by v, and set $S_{\text{parallel}}(v)$ of nodes that can neither reach nor be reached by v. On each set $S_i(v)$, we perform the aggregations using $h_i(\sum_{u \in S_i(v)} l_i(\mathbf{x}_u))$ where $\mathbf{x}_u$ are the node embeddings and h_i, l_i are multilayer perceptrons with trainable parameters as above. Finally, node v's embedding and the result from the three aggregations are concatenated as input to the subsequent policy network.

The above three steps define an end-to-end policy mapping from raw features associated with each op group to the device placement action. We train the policy network using a standard policy-gradient algorithm [18], with a timestep-based baseline [6] (see Appendix 5 for details).

Preliminary Experiments

In this section, we present our experimental setup and empirically evaluate Placeto. Our experiments answer: (1) how good are Placeto's placements in terms of execution time? (2) how well does Placeto generalize to unseen graphs, compared to prior approaches?

Setup. We consider the following well-known TensorFlow models as benchmark computational graphs in our evaluations: Inception-V3 [17], NASNet [20] and a 2-layer LSTM sequence-to-sequence model for language translation (NMT) [19]. As baselines, we compare against the RNN-based approach of [10] and a human-expert placement. The devices considered are two Tesla K80-GPUs; details on hardware and implementation can be found in Appendix A.1 and A.2.

Training. We train Placeto and the RNN-based approach [10] using a simulator that outputs run times of the TensorFlow models for different placements. We use a simulator since assessing run times (and hence rewards) is much faster on a simulator than on real devices. The simulator is used only for training; all the reported run time improvements are obtained by evaluating learned placements on real devices. We refer to Appendix A.3 for implementation details about the simulator.

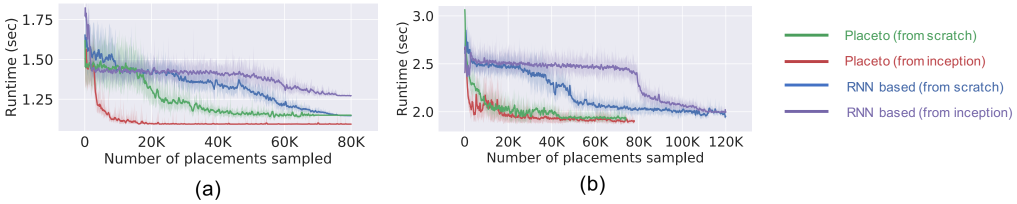

Figure 2: Learning curves for Placeto and RNN-based approach on (a) NASNet and (b) NMT models.

	Placement run time (sec)				Training time (# placements sampled)			
Model	Expert	RNN-based [10]	Placeto (scratch)	Placeto (transfer)	RNN-based [10]	Placeto (scratch)	Placeto (transfer)	Speedup factor
Inception-V3	1.27	1.21	1.20	—	11.2 K	3.7 K	—	—
NMT	2.00	1.52	1.52	1.57	84 K	20 K	3.9 K	**21×**
NASNet	0.86	0.84	0.83	0.84	76 K	28.8 K	12.2 K	**6×**

Table 1: Running times of placements found by Placeto compared with RNN-based approach [10] and human-expert baseline. The number of measurements needed to find the best placements for each of these approaches are also shown (K stands for kilo). Reported runtimes are measured using real devices.

Performance. Table 1 summarizes run times of the best placement found by each scheme. For all considered graphs, Placeto is able to rival or outperform the best comparing scheme. Placeto also finds the best placement much faster than the RNN-based approach. As shown in Table 1, Figures 2a and 2b, Placeto (shown in green) is able to achieve the same eventual performance with $3\times, 4.2\times$ and $2.6\times$ fewer number of training episodes compared to the RNN-based approach for Inception-V3, NMT and NASNet respectively. This shows the advantage of Placeto's simpler policy representation; it is easier to learn a policy to incrementally improve placements, than to learn a policy that decides placements for all nodes in one shot.

Generalizability. We evaluate the generalizability of each placement scheme by checking how well it transfers a learned placement policy to different TensorFlow models. For each placement scheme, we compare how fast it discovers good placements for a model when it is trained (1) starting from a random parameter initialization and, (2) starting from policy parameters learned for Inception-V3. To make the RNN-based approach dimensionally compatible across different graphs, we remove the graph-size dependent adjacency list feature (and associated parameters) from nodes in this scheme for the transfer experiment.[1] As shown in Figures 2a and 2b, the RNN-based approach converges with nearly identical rate when trained from scratch or from the other model's initialization. In contrast, Placeto discovers the best placement on NMT and NASNet $21\times$ and $6\times$ faster (Table 1). The RNN-based approach does not generalize because the placement policy is bound to node labels and traversal order on the graph.

Conclusion and Future Work

We presented Placeto, an RL-based approach for finding device placements to minimize training time of deep-learning models. By structuring the policy decisions as incremental placement improvement steps, and using graph embeddings to encode graph structure, Placeto is able to train efficiently and learns policies that generalize to unseen graphs. We will open-source Placeto and our simulator code.

We propose the following future directions for Placeto. Our generalization experiments (§3) have transferred a policy learned from a single model (Inception-V3) to other graphs. Using a mix of models with diverse graph structures during training, Placeto may exhibit better generalizability (e.g., transfer the learned policy to other graphs with zero or few-shot fine-tuning). Also, the placement problem is more challenging and can potentially lead to larger gains when considering larger graphs (e.g., NMT with 8 layers), larger batch sizes, and more heterogeneous devices. Lastly, Placeto currently relies on a manual grouping procedure to consolidate ops into groups. Inspired by the hierarchical placement approach from Mirhoseini et al. [9], we plan on extending Placeto to jointly learn grouping and placement policies in future work.

[1] While using graph adjacency information leads to better placements, the learned policy from the RNN-based approach is not compatible with a graph of different size. We found that removing these features still leads to reasonable runtimes while allowing transferability.

Acknowledgements

This work was funded by an AWS Machine Learning Research Award, the NSF grants CNS-1751009, CNS-1617702, a Google Faculty Research Award and Cloudlab [14].

References

[1] Ec2 instance types. https://aws.amazon.com/ec2/instance-types/, 2018. Accessed: 2018-10-19.

[2] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[3] P. W. Battaglia et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[4] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[5] Y. Gao, L. Chen, and B. Li. Spotlight: Optimizing device placement for training deep neural networks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1676–1684, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[6] E. Greensmith, P. L. Bartlett, and J. Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530, 2004.

[7] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[8] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963*, 2018.

[9] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[10] A. Mirhoseini, H. Pham, Q. Le, M. Norouzi, S. Bengio, B. Steiner, Y. Zhou, N. Kumar, R. Larsen, and J. Dean. Device placement optimization with reinforcement learning. 2017.

[11] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[12] A. Y. Ng, D. Harada, and S. J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[13] F. Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In T. P. A.-M. Kermarrec, L. Bougé, editor, *EuroPar*, volume 4641 of *Lecture Notes in Computer Science*, pages 195–204, Rennes, France, Aug. 2007. Springer.

[14] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[15] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[16] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[17] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[18] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[19] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *ArXiv e-prints*, 2016.

[20] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017.

Appendices

Implementation Details

Devices and Software

For all of our experiments, we use tensorflow r1.9 with a p2.8xlarge instance from AWS EC2 [1]. This instance is equipped with a Xeon E5-2686 broadwell processor and 8 Nvidia Tesla K80 GPUs.

Models

We evaluate our approach on the following popular deep-learning models from Computer Vision and NLP tasks:

Computer vision. We evaluate our approach on Inception-V3 [17] and NASNet [20] models which are widely used for various computer vision tasks. For each model we use a batch size of 64.

NLP. We also evaluate on a 2-layer RNNbased sequence-to-sequence model built for language translation which is known to be computationally expensive in training and inference [19]. We unroll the encoder and decoder networks to 40 steps, and use a batch size of 256.

Prior works [10, 9, 5] report significant possibilities of improvements in runtimes for several of the above models when placed over multiple GPUs.

Simulator

Over the course of training, runtimes for thousands of sampled placements need to be determined before a policy can be trained to converge to a good placement. Since it is costly to execute the placements on real hardware and measure the elapsed time for one batch of gradient descent [10, 9], we built a simulator that can quickly predict the runtime of any given placement for a given device configuration. For any given model to place, our simulator first profiles each operation in its computational graph by measuring the time it takes to run it on the available devices. We model the communication cost between devices as linearly proportional to the size of intermediate data flow across ops.

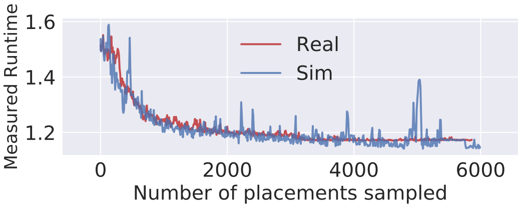

Figure 3: RNN-based approach exhibits near identical learning curve when reward signal is from a simulator or directly from measurements on real machines.

As a result, an RL-based scheme trained with the simulator exhibits nearly identical run times compared to training directly on the actual system. We demonstrate this by comparing the run times in the learning curves of a RNN-based approach [10] on the real hardware and our simulator (Figure 3).

Training details

Here, we describe training details for Placeto and RNN-based model. Unless otherwise specified, we use the same described methodology for setting the hyperparameters for both of these approaches.

Entropy. We add an entropy term in the loss function as a way to encourage exploration. We tune the entropy factor seperately for Placeto and RNN-based model so that the exploration starts off high and decays gradually to a low value towards the final training episodes.

Optimization. We tune the initial learning rate for each of the models that we report the results on. For each model, we decay the learning rate linearly to smooth convergence. We use Adam's optimizer to update our policy weights.

Workers. We use 8 worker threads and a master coordinator which also serves as a parameter server. At the beginning of every episode, each worker synchronizes its policy weights with the parameter server. Each worker then independently performs an episode rollout and collects the rewards for its sampled placement. It then computes the gradients of reinforce loss function with respect to all the policy parameters. All the workers send their respective gradients to the parameter server which sums them up and updates the parameters to be used for the next episode.

Baselines. For Placeto, we use a seperate moving average baseline for each stage of an episode. The baseline for time step t is the average of cumulative rewards at step t, of the past k episodes. We tune k for each model that we place.

For RNN-based approach, we use baseline as described in Mirhoseini et al. [10].

Neural Network Architecture For Placeto, we use single layer feed-forward networks during message passing and aggregation steps with the same number of hidden units as the input dimension. We feed the outputs of the aggregator into a two layer feed-forward neural network with softmax output layer. We use ReLU as our default activation function.

For the RNN-based approach, we use a bi-directional RNNwith a hidden size of 512.