EAGLE: Expedited Device Placement with Automatic Grouping for Large Models

Hao Lan*, Li Chen†, Baochun Li*
*University of Toronto, †University of Louisiana at Lafayette
hao.lan@mail.utoronto.ca, li.chen@louisiana.edu, bli@ece.toronto.edu

***Abstract*—Advanced deep neural networks with large sizes are usually trained on a mixture of devices, including multiple CPUs and GPUs. The model training speed and efficiency are drastically impacted by the placement of operations on devices. To identify the optimal device placement, the state-of-the-art method is based on reinforcement learning with a hierarchical model, which partitions the operations into groups and then assigns each group to specific devices. However, due to the additional dimension of grouping decisions coupled with the placement, the reinforcement learning efficiency is greatly reduced. With modern neural networks growing in size and complexity, the issue of low efficiency and high cost in device placement is further aggravated. In this paper, we propose our design of *EAGLE* (Expedited Automatic Grouping for Large modEls), which integrates automatic grouping into reinforcement learning-based placement in an optimal way, to achieve the best possible training time performance for very large models. An extra RNN is introduced to transform parameters of the grouper into inputs of the placer, linking the originally separated parts together. Further optimizations have also been made in the network inputs. We have deployed and extensively evaluated *EAGLE* on Inception-V3, GNMT and BERT benchmarks. Compared with the state-of-the-art, the performance achieved by our design, measured by the per-step time with the resulted placement, is 2.7% and 18.7% better for GNMT and BERT, respectively. For Inception-V3, our design achieves the fastest speed in discovering the optimal placement.**

***Index Terms*—device placement, reinforcement learning, neural networks**

I. Introduction

Deep neural networks (DNNs) have gained significant popularity in solving complex tasks such as image classification, speech recognition, and game playing. With ever increasing sizes and complexity, DNNs require an extensive amount of computational resources for training, and it becomes typical to leverage a heterogeneous mix of both CPU and GPU devices [1]–[3]. In such a distributed and heterogeneous training environment, *device placement* — the mapping of operations in a DNN to devices — plays a crucial rule in the training time performance.

Mirhoseini, *et al.* [4], [5] proposed to use reinforcement learning (RL) to find the best placement in order to reduce the time needed to train a neural network. Having identified the slow speed of convergence, Gao, *et al.* [6], [7] developed a joint learning algorithm that combines cross-entropy minimization and proximal policy optimization (PPO) to speed up convergence for the RL agent. The advantages of RL over human experts in device placement have been seen in these works: after about 12 to 27 hours of training, RL agents can find better placements than human experts for training neural networks such as Inception-V3 and NMT.

With the recent advances in artificial intelligence, the increasing size and complexity of newly developed DNN models bring significant challenges in device placement. For example, the BERT model, the state-of-the-art in natural language processing with over 10,000 operations, requires more than 16 GB of memory for its training even with a batch size of 1, which cannot be fitted into most of the GPUs [8]. In this context, existing approaches are no longer capable of optimizing the placement for such extremely large models: as the number of operations increases, it becomes impractical to group operations by human experts manually [4], [7]. To avoid manual grouping, Mirhoseini *et al.* [5] designed a hierarchical model which leverages a feed-forward neural network to group operations automatically and places these groups by a sequence-to-sequence neural network. However, training this hierarchical model is non-trivial since it involves two neural networks to be updated simultaneously.

In this paper, we propose our design of *EAGLE* (Expedited Automatic Grouping for Large modEls), a reinforcement learning agent that enables automatic grouping of operations and adopts a sample efficient training algorithm to find a better placement in a shorter period of time than all the existing works. Our design is driven by a comprehensive experimental investigation on an extensive array of RL training algorithms and deep neural network models of the RL agent for the device placement problem. With an in-depth analysis of their pros and cons, we design our agent with the favorable integration of grouping and placing, and the best combination of the algorithm and the model. For example, we reconstruct the state vectors to be fed into the RL agent, which makes the agent better understand the computational graph of the neural network.

We have deployed *EAGLE* on 4 GPU machines in the Compute Canada platform to find the best placement of influential deep neural networks, including Inception-V3, NMT, and BERT. Under a challenging scenario when a very large model is to be placed, *EAGLE* is able to find the best placement

The co-authors would like to acknowledge the gracious research support from Huawei Technologies Canada Co., Ltd., as well as the grant from Louisiana Board of Regents under the contract LEQSF(2019-22)-RD-A-21.

among all the baselines while existing works even fail to find a placement better than human experts. Within four hours of learning, the best placement discovered by *EAGLE* reduces the training time of the GNMT model by 17.0% compared to human expert placement, and it is 2.5% better than the placement found by Hierarchical Planner [5]. While Hierarchical Planner fails to learn how to place the BERT model, *EAGLE* finds a placement which is 18.7% better than the placement found by Post [7].

This paper makes three primary contributions. *First*, we have identified the urgent need and challenges of efficient device placement for training very large neural network models. *Second*, we have conducted a comprehensive evaluation and in-depth analysis of existing approaches in the design space, which motivate and inspire our design of *EAGLE*. *Third*, we have proposed a practical design of a reinforcement learning based device placement agent, which enables automatic grouping of enormous operations in a computational graph and realizes efficient learning of the optimal placement. *Finally*, we have implemented *EAGLE* for benchmark deep learning models, and our experimental results have demonstrated that *EAGLE* outperforms existing baselines in discovering better placement and learning with faster speed.

The remainder of this paper is organized as follows: We briefly present the background knowledge and the state-of-the-art in Sec. II, and identify the challenges and opportunities that motivate our design. In Sec. III, we present an overview of our design, covering the design space and our choices for the grouping methods, the placement strategies, as well as the hierarchical model architecture that integrates these components. We evaluate *EAGLE* with experiments over a set of baselines and benchmarks in Sec. IV. Finally, we conclude our paper in Sec. V.

II. Device Placement for Deep Learning

In this section, we briefly describe the training phase in deep learning, covering the widely known model for image classification and the latest complex model for natural language processing. For these computation-intensive and resource-demanding tasks in model training, we will go through the state-of-the-art approaches in distributed and parallel training, identity the challenges, and present opportunities that motivate our design.

A. Deep Neural Network Training

Training a deep neural network typically involves a massive amount of computation over thousands of operations.

With image classification as an example, the model to be trained is called a *convolutional neural network* (CNN), which is a layered network of nodes (operations) and edges (connections with weights). The training of such a model is in an iterative fashion, with each iteration consisting of a forward pass and a backward pass. In a forward pass, a batch of input examples (*i.e.*, raw pixels of images [3]) is fed into the input layer of the network, which would then be multiplied by a series of weight matrices through hidden layers to generate the output. In particular, the convolutional layer applies a convolution function and the fully connected layer applies a weighted summation over the values of the nodes connected to them at their respective prior levels. The outputs at the last layer, *i.e.*, the predicted label probabilities for an image in this example, are compared with the true label to generate a loss value that measures the prediction error. The loss values are used to compute gradients of connection weights and then update these weights, in a backward pass [9]. It is typical that training a neural network requires millions of iterations to gradually reduce the loss value below a threshold, which can take days or weeks.

For a more complex model as BERT [8], both the large number of parameters and sophisticated training approaches make its training time-consuming. BERT-Large has 24 transformer layers, 16 attention heads, and 345 million parameters, which is the largest model of its kind. Even a small version of BERT, BERT-Base, has 12 transformer layers, 12 attention heads, and 110 million parameters. In addition to its large scale, BERT's bidirectional approach also converges slower than left-to-right approaches. As a result, BERT-Large (BERT-Base) requires 4 days of training with 16 (4) Cloud TPUs.

B. Distributed Deep Learning

With the increasing size of training datasets and the increasing complexity of deep neural network architectures, the computation and memory demands of deep learning grow significantly, requiring computing clusters to exploit the concurrency for high-performance training.

Data Parallel Training. In traditional machine learning clusters, data parallelism is extensively employed to achieve high scalability, *e.g.*, with the parameter server architecture [10]. In data parallel training, each worker device maintains a copy of the complete model with all the operations running to train its own partition of input data. To ensure a consistent copy of the model across all the workers, the parameters of each model copy are periodically synchronized through either the parameter server or allreduce communication. However, with the increasing size and complexity of deep neural networks, it becomes difficult to fit a copy of the complete model into GPU memory. Although it is possible to reduce the activation memory of training by using techniques such as gradient checkpointing, frequently storing and loading intermediate tensors will slow down the training.

Model Parallel Training. To meet the demanding computation requirements of deep learning, it recently becomes typical to train deep neural networks in a heterogeneous cluster, consisting of a mixture of CPU and GPU devices [1]–[3]. As the memory size is limited in a GPU device and the model size keeps increasing in today's deep neural network, model parallelism is widely used to partition a large model across multiple devices, when the model cannot fit into the memory of a single device.

In this setting, machine learning practitioners are given the flexibility to customize the mapping between devices and operations in their neural network models. For example, in

TensorFlow [11]), a user can specify the device for each operation, which will be executed accordingly on the device during the training phase. Intuitively, *device placement*, *i.e.,* how operations are assigned to devices, significantly influences the training time. However, it is non-trivial to find the optimal placement for neural network models. For example, for the placement of 1000 operations on four GPUs and one CPU, there are 5^{1000} possible assignments in total. The growing depth and size of modern neural networks exaggerate the problem of finding the best placement among the astronomical number of possibilities.

C. Device Placement with Reinforcement Learning

Essentially, the device placement problem can be described as a graph partitioning problem, where the operations are partitioned (and assigned) to a few different groups (devices). Although there are many well-studied algorithms for graph partitioning problems, such as the Scotch optimizer [12], a recent study has shown that these algorithms yield disappointing results in device placement settings. The reason is that their partitioning strategies are extensively tuned and not flexible enough when dealing with TensorFlow computational graphs.

To address the challenge, Mirhoseini *et al.* [4] proposed to train a neural network with reinforcement learning (RL) to learn how to place operations optimally. The main idea was to perform a series of experiments with the environment information taken into account, and to gradually learn an optimal placement where the operations are arranged towards an optimal communication. More specifically, the RL agent randomly generates a placement of a neural network. Given this placement, the environment, a real-world machine with a mixture of heterogeneous devices, measures the per-step runtime by training the neural network for several steps. The RL agent takes this per-step runtime as a reward signal to adjust its policy of generating future placement towards a better one with a shorter per-step runtime, until a best-performed one is eventually obtained.

Although this approach found a better placement than human experts, the training cost of the RL agent — taking between 12 to 27 hours with 80 to 160 4-GPUs machines — was prohibitively high [4]. Spotlight [6] proposed to use an advanced RL algorithm based on proximal policy optimization (PPO) [13] to improve the training efficiency. As a result, Spotlight significantly reduced the cost of learning optimal device placement, making it affordable for machine learning users to accelerate their neural network training.

As a further improvement, Post [7] used a joint training algorithm that combined the proximal policy optimization and cross-entropy minimization to accelerate the training of the RL agent. It made incremental policy improvements by updating the policy network with PPO every several sampling periods. After collecting a certain number of samples, Post solved a cross-entropy minimization problem to achieve a global and aggressive policy improvement. With such a joint training algorithm, it achieved faster convergence and obtained a better placement for some neural network models compared to Mirhoseini *et al.*'s work [4].

Instead of placing all the operations in one shot, Placeto [14] proposed to place only one group of operations and evaluate the placement for every single change, so that the reward can directly reflect the changes that have been made in each step. However, this approach required an extremely large number of steps to train, as it needed hundreds of steps to place the entire model. Hence, they used a simulator to evaluate the placements, rather than collecting measurements from real devices.

The state-of-the-art works had one characteristic in common: they all grouped the operations of neural networks before placing them. This is because they had to reduce the action space of the RL agent, otherwise the difficulty of training will be unacceptable. Such a grouping process was done manually before Mirhoseini *et al.* proposed a hierarchical model, called Hierarchical Planner, consisting of two neural networks to partition the operations automatically [5], as shown in Fig. 1. The grouper — a two-layer feed-forward neural network — took all the operations as input and partitioned them into a fixed number of groups. Then, the embeddings of these groups were generated and fed into the placer, which is a sequence-to-sequence neural network. The grouper and placer were trained jointly with the RL algorithm. Therefore, the hierarchical model itself can learn how to group and place the operations at the same time. This two-level hierarchical design allowed the agent to directly take computational graphs of the ML jobs as input, rather than manually sliced operation groups. However, since the grouper was initialized with random parameters, the grouping result was far from optimum at the beginning. Moreover, the dynamics of the grouping result during training made it even harder to train the agent.

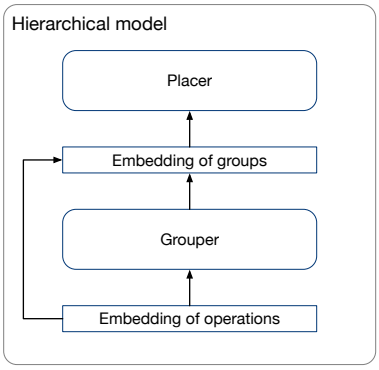


Fig. 1: An illustration of the hierarchical model.

Besides grouping operations, Zhou *et al.* [15] proposed to use a graph encoder to learn a representation for each operation and place all operations by a modified transformer-XL model directly. While this approach granted the finest granularity of placing operations to the placer, it increased the training difficulty.

Some relate works tried to speed up the training of the neural networks in a distributed computing environment. Tic-

Tac [16] accelerated a distributed deep learning system by communication scheduling. It consisted of two heuristics for efficient scheduling and improved iteration throughput by 20%. Priority-based Parameter Propagation (P3) [17] also improved the training performance by better utilizing the available network bandwidth. It splits the layers into smaller slices and synchronizes them based on their priority independently. PipeDream [18] combined traditional data parallelism with model parallelism enhanced with pipelining. It automatically partitioned a neural network and used pipelined parallelism across multiple machines. Different from these works, We focus on model parallelism across multiple devices on a single phyiscla machine.

D. Challenges and Opportunities

Due to the slow convergence speed of reinforcement learning, the cost of finding the best placement is prohibitively high, even for a relatively small neural network model such as Inception-V3. When it comes to recent neural network models with very large sizes and complicated structures, such as BERT, the disappointing learning speed of existing works can hardly tackle the challenges in more practical settings. More specifically, we have identified the following challenges and opportunities to motivate and inspire our design.

Challenge 1: Manual grouping is neither optimal nor scalable. Manual grouping requires an in-depth understanding of the model architecture and running environment to make accurate estimation and judicious decisions. This is inherently difficult, given the number of operations, the complexity of deep learning models, and the heterogeneity of running environments that jointly lead to a huge design space.

Challenge 2: Learning based grouping exaggerates the slow convergence of placement learning. Grouping and placing are inherently coupled, and they jointly determine the final training time performance. Particularly, given each possible way of grouping, there is a large space of placements to be explored. Even with fixed grouping, placement using RL takes time. Such a two-dimensional learning space will incur significant overhead and become impractical, as even one-dimensional learning of placement with fixed groups is already slow.

Challenge 3: A unified end-to-end device placement framework remains largely unexplored. The device placement problem has mostly been addressed by RL approaches for the single dimension of placing, taking fixed operation groups as input. To the best of our knowledge, there are only two existing works proposed to operate in the end-to-end fashion, which reads the computation graph as input and predict the placement by two joint neural networks [5], [15]. This category of solutions, without a delicate design, may easily suffer from slow convergence and heavy overhead, due to an additional neural network introduced to be trained jointly. In a unified learning framework with the computation graph as input and device placement as output, a rich set of design choices remains unexplored.

Faced with these challenges, we with to explore the opportunities of designing a RL agent that integrates automatic grouping in a light-weighted fashion with highly efficient placement learning. In the following section, we will present our design choices of the model architecture and training algorithm in such a unified learning framework.

III. Design

We present our design of *EAGLE* from two aspects: the model used in the RL agent and the training algorithm for model updates.

A. Overview of the Hierarchical Model

It is a common practice in existing works [4], [6], [7], [14] to partition the operations in the computational graph of a machine learning job into small groups, and then use a neural network to decide the device for each group of operations to be placed on. The rationale behind this design is that the number of operations in today's deep neural networks is huge, which implies that the action space of directly placing these operations becomes enormous, and the agent can hardly find the best placement. Instead of manual grouping, Mirhoseini *et al.* [5] introduced a hierarchical model which uses a feed-forward neural network, called a grouper, to automatically group the operations.

Our design of *EAGLE* is driven by the need of striking a balance between the sophistication of grouping and the efficiency of training. Manual grouping requires an expert with a deep understanding of the model of a machine learning job, who makes judicious grouping decisions according to the co-location information within the computational graph and the potential resource bottleneck in the model. This is not a general solution for a wide variety of modern neural network models, especially when these models evolve rapidly with increasing complexities in their structures.

Hence, our design space is within the scope of automatic grouping, and our next concern is the choice between a heuristic-based approach following the traditional convention and a learning-based approach in the newly emerging proposals. Compared with manual grouping that entirely relies on human intelligence and knowledge, a learning-based approach, in contrast, relies completely on machine intelligence. However, such a benefit comes with the cost of introducing more complexities in training, especially since the grouper and the placer are tightly coupled with each other and are to be jointly trained. To mitigate the difficulty of training, we re-investigate the well-studied research area of graph partitioning, with the hope of achieving a favorable tradeoff between the need for automation and computational requirements.

B. Grouper Design: Model-based vs. Heuristic-based

We study the following two graph partitioning heuristics: (1) the asynchronous fluid communities algorithm, a community partitioning algorithm implemented in the Python `Networkx` package [19]; and (2) METIS [20], one of the popular graph partitioning algorithms. To benchmark these algorithms in the

context of device placement, we have reproduced Hierarchical Planner [5] based on the source code in TensorFlow grappler [21], and replaced the feed-forward neural network with these graph partitioning heuristics. Unlike machine learning methods that can utilize multi-dimensional features of nodes, graph partitioning heuristics usually only focus on the weights of nodes and edges in the graph. Given a neural network model to be grouped and placed, we extract features from its computation graph and build an identical graph, where the edges are the connection between operations, and the weight of each edge is the amount of data needed to be transmitted from the source to the destination operation. These features represent the communication cost between operations. The heuristics will partition the graph by solving the min-cut problem; and as a result, the total communication cost between these groups will be minimized. With the grouping output from the heuristics, the placer will be trained in the RL agent to search for optimal placement for the operation groups.

TABLE I: Per-step time (in seconds) of placements found by the hierarchical model with different groupers.

Models	Feed-forward	METIS	Networkx
Inception-V3	**0.067**	0.071	0.072
GNMT	**1.418**	1.537	2.041
BERT	**5.534**	7.526	7.584

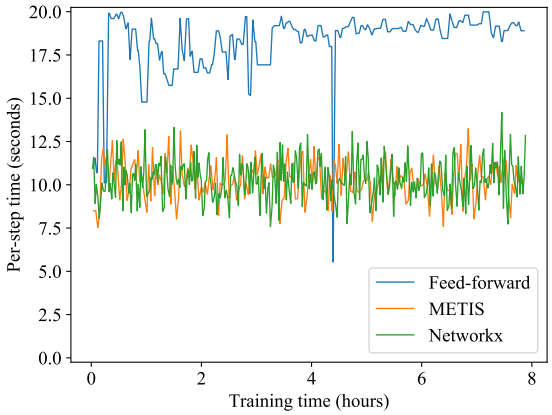


Fig. 2: Per-step time of the placement for BERT, found by the hierarchical model with different groupers during the training process.

We evaluate and present the performance of placements for three models found by different strategies in Table I. These models are included in the TensorFlow framework, among which the Inception-V3 [22] and GNMT [23] models are enabled with the default placement across multiple devices, and the BERT model is not.

As shown in Table I, the best placement (with the shortest per-step time) identified by the agent with the feed-forward neural networks outperforms the agents with heuristic-based groupers (METIS and Networkx). However, we found that the hierarchical model with neural network-based grouper converged to a local optimum rather than the best placement. As shown in Fig. 2, although it found a better placement for BERT during the training process, it eventually converged to a worse point than the two heuristics. This is because when the model grows larger and more complex as BERT, the additional dimension of variables in learning-based grouping incurs significantly more training overhead and makes it difficult for the agent to learn from good placements. Hence, efficient training is highly desired for optimally placing large models. If the hierarchical model with the feed-forward neural networks is fully trained, it should be able to group and place the operations in the right way, outperforming heuristic-based grouping to a large extent. With such observation and insight, we decide to use feed-forward neural networks as the grouper of *EAGLE*, and try to improve the training efficiency by designing a simpler placer and by using advanced training algorithms.

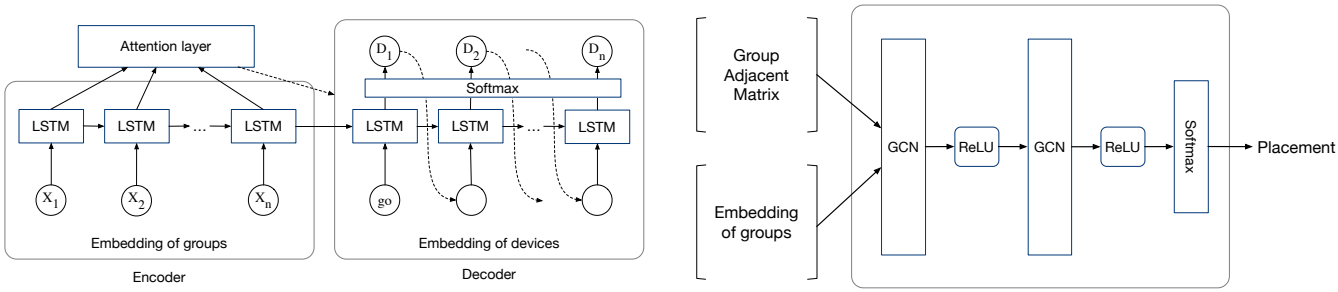
C. Placer Design: A Deep Dive

With respect to the placer, we compare two neural networks, a sequence-to-sequence neural network [1] and a graph convolutional network (GCN) [24]. Both of these neural networks take the embedding (or features) of operation groups as input, but they generate placements in different ways. The sequence-to-sequence placer is a recurrent neural network that outputs the device for each group one by one. In contrast, the GCN placer outputs the hidden states of all groups at the same time, and then predicts the entire placement for them.

As shown in Fig. 3a, our sequence-to-sequence placer has an encoder and a decoder, each consisting of a long short-term memory (LSTM) layer. After the encoder encodes the information of the entire graph — a sequence of group embeddings $\{X_1, X_2, \ldots, X_n\}$ — into a fixed-length vector called hidden states, the decoder decodes this vector and predicts the device D_n of the group n to be placed on. A group embedding consists of three parts: the number of operations of each operation type in the group, the output shapes, and the adjacency information of the group. We generate these embeddings by aggregating the embedding of operations in the same group in the same way as Hierarchical Planner.

As illustrated in Fig. 3b, the GCN placer takes two inputs, the embedding of groups and an adjacency matrix. As the adjacency matrix already has adjacency information of groups, we removed the adjacency information in group embeddings. We use two graph convolutional layers with the ReLU activation function in the model, which finally outputs the policy through a softmax layer.

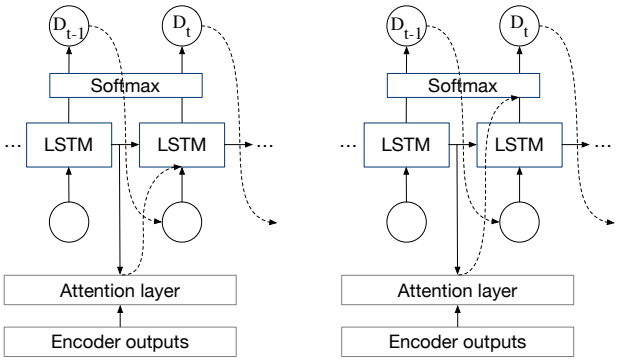
We found that there are two different methods of using the attention layer in the sequence-to-sequence neural network. As shown in Fig. 4, the context calculated by the attention layer can be combined before or after the RNN decoder, *i.e.*, the LSTM layer. In Fig. 4a, the LSTM takes the output of the attention layer, previous hidden states of the decoder and the embedding as inputs, and generates the next hidden states. The output is predicted by the softmax of the next hidden states.



(a) A sequence-to-sequence network model.

(b) A graph convolutional network model.

Fig. 3: Illustration of two different placer designs.



(a) before decoder

(b) after decoder

Fig. 4: Illustration of two different methods of applying attention mechanisms.

In Fig. 4b, the output of the attention layer is used after the decoder layer, and the prediction is based on the softmax over the outputs of the attention layer and decoder layer. Google's Hierarchical Planner uses the method presented in Fig. 4b to apply the attention layer. For the attention mechanism, we adopt the mechanism proposed by Bahdanau *et al.* [2], which calculates the context vector based on encoder outputs and previous hidden states of the decoder.

TABLE II: Per-step time (in seconds) of placements found by the agent with METIS grouper and different placers.

Models	Seq2Seq(before)	Seq2Seq(after)	GCN
Inception-V3	**0.067**	**0.067**	0.072
GNMT	1.440	**1.418**	2.040
BERT	**4.120**	5.534	7.214

We evaluate both versions of the sequence-to-sequence placer and the GCN placer with three benchmarks. To eliminate the influence of the grouper, we train these three placers with a fixed grouping, which is generated by the METIS grouper. From the experimental results shown in Table II, we found that the sequence-to-sequence placer outperforms the GCN placer in all the benchmarks. The reason is that the GCN placer makes decisions for each group independently while the sequence-to-sequence placer predicts the device of a group based on previous decisions. Among two versions of sequence-to-sequence placer, although the placement of the GNMT model found by the before version is slightly worse compared to the after version, it finds a much better placement of the BERT model. So we use the before version of sequence-to-sequence placer in *EAGLE* to better address large models.

D. Advanced Training Algorithm

The training algorithm plays an important role in the learning speed and efficiency of a RL agent. In this section, we examine an array of training algorithms in the context of placing very large models, including the REINFORCE, proximal policy optimization [13] and Post [7].

The REINFORCE algorithm is the most basic and popular training algorithm for reinforcement learning. It is simple and easy to be used to solve a wide variety of practical problems, but it may not be the best choice for device placement. Compared with the environments where the interactions and rewards are easily obtained such as in video game playing, the cost of interactions in device placement is not cheap. For example, the average time of evaluating a random placement with 10 steps of the NMT model is about 1 minute. As a result, the sampling rate of the device placement environment is very slow, which makes it crucial to leverage these samples in a better way. To achieve a higher sampling efficiency, we use a more advanced policy gradient method, proximal policy optimization, which performs comparably or better than state-of-the-art approaches while being much simpler to implement and tune. In this paper, we use the clip version of PPO which is the best one reported in [13]. The objective function is shown in Eq. (3),

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \tag{1}$$

$$L^{\text{TRPO}}(\theta) = \mathbb{E}[r(\theta)\hat{A}_t(s,a)] \tag{2}$$

$$\begin{aligned} L^{\text{CLIP}}(\theta) = \mathbb{E}[&\min(r(\theta)\hat{A}_t(s,a), \\ &\text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t(s,a))] \end{aligned} \tag{3}$$

where π_θ is the policy based on current parameters θ, θ_{old} is the parameters before updating, and $\hat{A}_t$ is the estimated advantage for action taken at time step t. To avoid an excessively large

policy update, PPO uses clipped surrogate objective over trust region policy optimization (TRPO) in Eq 2. As a result, it enables multiple epochs of minibatch updates without moving too far away from the old policy. Here, ϵ is a hyperparameter for adjusting the clip region.

Instead of directly updating the policy network with reward, advanced reinforcement learning algorithms perform better in an A2C (Advantage Actor-Critic) [25] fashion — the agent uses a value network to predict the value of each action and estimates the advantage based on the gap between the value and the actual reward. However, in our attempt at proximal policy optimization in an A2C fashion, the value network does not have enough samples to be trained and may yield inaccurate estimations. The inaccuracy will lead to the policy network updating towards a wrong direction, which aggravates the difficulty of convergence. To solve this problem, we use the exponential moving average of rewards as a baseline and calculate the advantages by subtracting the baseline from rewards. As shown in Eq. (4), we use the negative square root of the per-step time of placements as the reward,

$$\begin{aligned} R_t &= -\sqrt{r_t} \\ B_t &= \text{ExpMovAvg}(R_t) \\ \hat{A}_t &= R_t - B_t \end{aligned} \tag{4}$$

where r_t is the per-step time of the placement sampled at time step t.

Apart from algorithms of REINFORCE and PPO, we also consider the joint training algorithm proposed in Post, which combines proximal policy optimization and cross-entropy minimization. This algorithm updates the agent with proximal policy optimization every few samples, which is exactly the same as original proximal policy optimization. After collecting a large number of samples, it picks the top K samples as elites and updates the agent with them. By doing this, the agent is more likely to probe around the good placements previously found.

To evaluate the algorithms, we train *EAGLE* with these three algorithms and compare the per-step time of best placements they found for three different models.

TABLE III: Per-step time (in seconds) of placements found by *EAGLE* trained with three different algorithms.

Models	REINFORCE	PPO	PPO+CE
Inception-V3	**0.067**	**0.067**	**0.067**
GNMT	2.216	**1.379**	1.507
BERT	2.425	**2.287**	2.488

As presented in Table III, proximal policy optimization is the best training algorithm for our model. It outperforms all other algorithms both in final results and convergence speed. Compared to REINFORCE and PPO joint with cross-entropy minimization (represented as PPO+CE in Table III), the per-step time of placements found by the agent trained with PPO is shorter, while the training time is also the shortest. PPO joint with cross-entropy minimization does not work well when training with *EAGLE*. It finds a better placement for the GNMT model compared to PPO, but it falls at a local optimum when optimizing the placement for BERT.

IV. Experiments

In this section, we evaluate *EAGLE* with three widely-used deep neural networks of different architecture and size as benchmarks. And we compare our results to four baselines, including two pre-defined placements and two RL-based approaches. To show why *EAGLE* can find a better placement than other approaches, we also examine the training process of all RL-based approaches.

A. Benchmarks

We choose three widely-used deep neural network models for computer vision, neural machine translation, and language representation learning. The size of models also spans from small, large to very large.

- **Inception-V3**, the third edition of Google's Inception Convolutional Neural Networks, which has been widely used in computer vision tasks, such as recognition, classification and feature extraction [22]. It is one of the benchmarks used in the evaluation of the state-of-the-art RL-based approaches. With a relatively small size, Inception-V3 can easily fit into a single GPU. We use it as a base case to evaluate the ability of an agent to find the best placement. The batch size of the model is set to 1.
- **GNMT**, a neural machine translation model proposed by Google [23] for automated translation. It has three variations with a different number of LSTM layers: 2-layer, 4-layer, and 8-layer. We use the 4-layer version with an attention layer, where each LSTM layer has 256 hidden units. The sequence length is limited within 20 to 50. To make it more challenging for the RL agent to find the best placement, we increase the batch size of the model from 128 to 256, such that it cannot fit into a single GPU. All the other settings are left as default.
- **BERT**, a novel language representation model proposed recently, with a large number of operations and a complex design [8]. It is supposed to be trained on a Cloud TPU with 64GB of device RAM, while typical GPUs only have 12GB to 16GB. Even with a batch size of 1, the BERT-Large model cannot fit on a 12GB GPU. So we use a smaller version of BERT model, BERT-Base, with a max sequence length of 384 and a batch size of 24. With this setting, the model still cannot fit into a single GPU but is able to be trained by placing its operations across four GPUs.

B. Baselines

To demonstrate the improvement achieved by *EAGLE*, we compare *EAGLE* with two pre-defined placements and two state-of-the-art RL-based approaches:

- **Single GPU.** As the name suggested, this baseline tries to place all operations on a single GPU. For the operations that are incompatible with GPU, such as embedding

lookup, we place them on CPU devices. This baseline is only valid for models that can fit into a single GPU, *e.g.* the Inception-V3 in our benchmarks. The large models will trigger an Out-Of-Memory error when training with a single GPU.

- **Human Expert.** We use the pre-defined placements from open source libraries of three models. For Inception-V3, we use the pre-defined placement in TensorFlow-Slim library [26], which places most of the operations on the same GPU and the rest on CPU. For GNMT (Google's NMT) [27], it places each LSTM layer, attention layer and softmax layer on a separate device while using multiple GPUs. For BERT, we use the implementation provided by Google [28]. however, it does not have pre-defined placement for multi-GPU training with model parallelism.
- **Hierarchical Planner.** Mirhoseini *et al.* [5] proposed a hierarchical model consists of a grouper and a placer, which are trained jointly to learn the policy for grouping and placing operations in a neural network. This hierarchical model avoids manual grouping of operations and optimizes the placement of a neural network in an end-to-end fashion. It is one of the state-of-the-art works for solving the device placement problem.
- **Post.** Post [7] uses a joint learning algorithm, which combines proximal policy optimization and cross-entropy, to train a simple neural network for device placement problem. It greatly improves the sample efficiency and achieves an impressive reduction both on training time and the per-step time of final placement.

C. Experimental Setups

Following the convention of reinforcement learning-based approaches, we train *EAGLE* for the device placement of three benchmarks, with the following settings:

- **Environment.** The environment in our training is a physical machine, which has 4 NVIDIA P100 Pascal GPUs, 2 Intel E5-2650 v4 Broadwell @ 2.2GHz CPUs and 125GB memory. We use Python 3.6 and TensorFlow r1.12 for running the deep neural network models.
- **Agent architecture.** As mentioned in Section 3, *EAGLE* employs a hierarchical design for grouper and placer. We explored a set of grouper designs and found that a two-layer feed-forward neural network with 64 hidden units is the best. The number of groups is set to 256 in our experiments. The placer is a sequence-to-sequence model with an attention layer. It has a bi-directional LSTM layer as the encoder and a uni-directional LSTM layer as the decoder. Both of these two LSTM layers have a hidden size of 512. The attention mechanism we used is context-based input attention proposed by [2]. The attention score is applied before feeding to the decoder. The agent is implemented with Python 3.6 and PyTorch v1.0.
- **Training algorithm.** We use two different algorithms to train *EAGLE*, PPO and PPO joint with cross-entropy minimization. This is because when the deep neural network to be placed is complex, such as BERT, it is hard for the agent to find the optimal placement among the enormous action space. In this case, cross-entropy minimization will let the agent explore more around the elite placements found during the training. However, this may also lead to the agent falling into a local optimum. So we evaluate *EAGLE* with both training algorithms. For the PPO algorithm, we collect 10 placements as a mini-batch and update the parameters of agent 4 times for each mini-batch. The clip ratio, ϵ, is 0.3 and the coefficient of entropy is 0.01. For the PPO joint with cross-entropy minimization, we use the same hyperparameters for the PPO part, and set the interval of two cross-entropy minimization updates to 50 placements. The number of elites is 5, which means it picks the top 5 placements from all sampled placements. We use Adam optimizer to train our agent with a learning rate of 0.01, and clip gradients by norm at a threshold of 1.0.
- **Placement evaluation.** We measure the per-step time of placement by running it on a physical machine (environment). In the training phase, we evaluate each placement sampled from the policy by running it for 15 steps, and take the average per-step time over the last 10 steps. This is because, whenever there is a new placement sampled, the environment needs to initialize the parameters on different devices based on the new placement and the first few steps will take longer time to finish. So we discard the first 5 warm-up steps and average the per-step time over the last 10 steps. After the training, we pick the best placement found by the agent and run it for 1,000 steps. The same as before, we discard the first 5 warm-up steps and average per-step time of the rest.

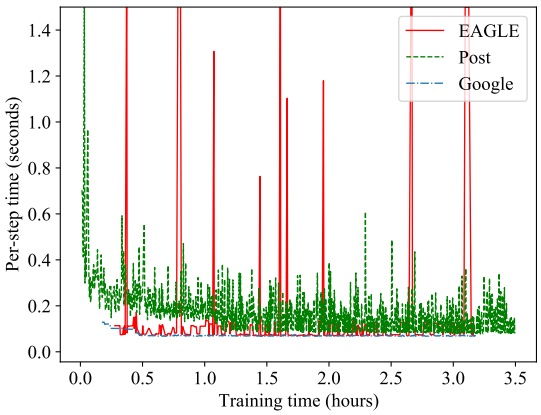


Fig. 5: Per-step time of the placement for Inception-V3 found by different approaches during the training process.

D. Results and Analysis

Figures 5 to 7 show the per-step time of placements found by three RL-based approaches for benchmarks.

TABLE IV: Per-step time (in seconds) of placements found by different approaches (lower is better). OOM stands for Out-Of-Memory.

Models	Single GPU	Human Experts	Hierarchical Planner	Post	EAGLE (PPO)	EAGLE (PPO+CE)
Inception-V3	0.071	0.071	**0.067**	**0.067**	**0.067**	**0.067**
GNMT	OOM	1.661	1.418	2.031	**1.379**	1.503
BERT	OOM	OOM	5.534	2.812	**2.287**	2.488

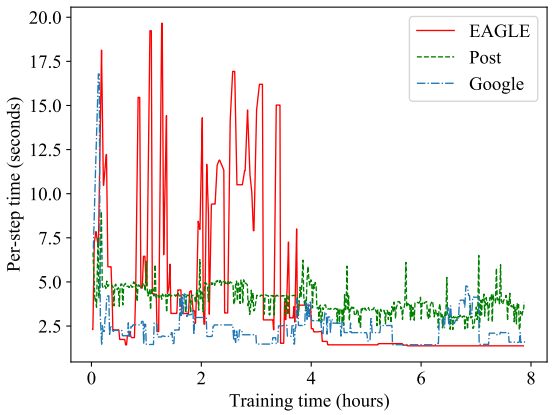


Fig. 6: Per-step time of the placement for GNMT found by different approaches during the training process.

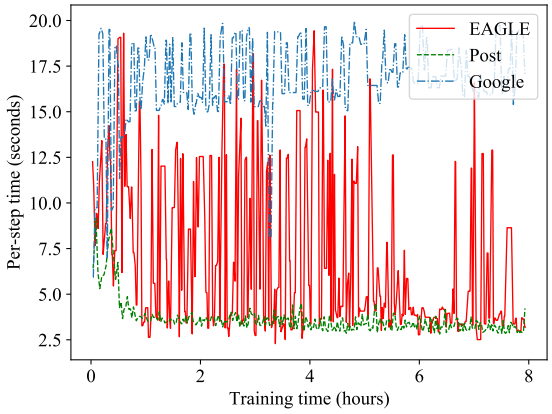


Fig. 7: Per-step time of the placement for BERT found by different approaches during the training process.

For the Inception-V3 model, we find that Google's Hierarchical Planner experiences a large number of invalid placements at the beginning of training. After learning with 200 placements, it starts to generate valid placements and tries to further optimize the placement. *EAGLE* and Post are better at avoiding the invalid placements. They only encounter very few invalid placements during the entire training process. All three approaches are able to find the optimal placement within 3.5 hours, and *EAGLE* is the fastest one.

For the GNMT model, all three approaches converge within 6 hours. Both Google's Hierarchical Planner and *EAGLE* find a good placement within 1 hour. Then, they keep exploring more placements and try to find a better placement. In this phase, *EAGLE* is more aggressive, the per-step time of placement fluctuates greatly over time. Post has a relatively bad start point. The per-step time of the best placement found in the first four hours is 5 seconds. Although it becomes better after four hours, it finally converges to a local optimum, where the per-step time of placement is around 4 seconds.

For the BERT model, Google's Hierarchical Planner fails to learn how to place it and converges to a bad point, where the per-step time of placements is around 17 seconds. Post converges fast and is stable. It finds a very good placement in the first hour and keeps improving the placement gradually for the rest of the training time. Similar to the previous two benchmarks, *EAGLE* always tries to explore as many placements as possible in the first few hours and becomes stable at the end of the training. As a result, *EAGLE* finds the best placement for the BERT model among these three baselines.

From the results, we have the following conclusions: In terms of convergence, Post is the most stable one. This is because that Post has a much simpler neural network compared to the other two RL-based approaches such that it can easily be fully trained. However, the simplicity of the neural network also means it may not be able to find the best placement for the model, which is exactly what happened when placing the GNMT model.

Final placements compared with state-of-the-arts: Table IV presents the per-step time of the best placement found by *EAGLE* and the two state-of-the-art works (Google's Hierarchical Planner and Post). *EAGLE* outperforms the other two RL-based approaches in all three benchmarks.

For the Inception-V3 model, all RL-based approaches find very similar placement – putting most operations on the same GPU device, similar to the pre-defined placements. This is most likely because the Inception-V3 model is too simple, only taking a few milliseconds to finish a step. As a result, the communication overhead of sending data across devices outweighs the benefits of leveraging multiple devices. We also notice that the best placement found by RL-based approaches is slightly faster than the pre-defined placements. The reason is that some operations are actually running faster on the CPU devices, and the RL-based approaches learned this knowledge during the training and finally generate a better placement than the pre-defined placements.

For the GNMT model, both Google's Hierarchical Planner and *EAGLE* found a better placement than the human experts,

which reduced the per-step time by 14.5% and 17.0%. In contrast, Post went into a local optimum and failed to find the best placement for GNMT.

For the BERT model, it requires more than 12GB of device RAM to train, which is not able to fit into a single GPU. And it also does not support running over multiple GPUs by default. Therefore, we only compare the placements found by the three RL-based approaches. All of them are able to find a valid placement for the BERT model. However, from the observation of the training process, we find that Google's Hierarchical Planner failed to learn how to place the BERT model and generated bad placements. Post and *EAGLE* performed well in this benchmark, finding good placements with per-step time below 3 seconds. The best placement is found by *EAGLE*. It reduces per-step time by 18.7% compared to the placement found by Post.

V. Conclusion

Recent years have witnessed the growing size and complexity of advanced neural network models. When training these large models across multiple computation devices, it becomes increasingly important to find an optimal mapping of operations to devices, so that the model training completes as fast as possible. This problem is challenging due to the massive amount of operations in large neural network models. In this paper, we have presented our design of *EAGLE*, a unified reinforcement learning framework to automatically partition a neural network model into groups and learn the best device placement for these groups. Particularly, our design is based on comprehensive analysis and experimental evaluation of a rich set of design choices for the model architecture and training algorithm. Finally, we have implemented and evaluated *EAGLE* with three benchmarks, *i.e.*, Inception-V3, GNMT and BERT. Compared with existing baselines, *EAGLE* has demonstrated its superiority in discovering better placements for large models of GNMT and BERT. For Inception-V3 which is relatively smaller, *EAGLE* is able to find the optimal placement with the fastest speed.

References

[1] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, 2014.

[2] D. Bahdanau, C. Kyunghyun, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. International Conference on Learning Representations*, 2015.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Computer Vision and Pattern Recognition (CVPR)*, 2016.

[4] A. Mirhoseini, H. Pham, Q. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning*, 2017.

[5] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *Proc. Int'l Conference on Learning Representations (ICLR)*, 2018.

[6] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *International Conference on Machine Learning*, 2018.

[7] ——, "Post: Device placement with cross-entropy minimization and proximal policy optimization," in *Advances in Neural Information Processing Systems*, 2018, pp. 9971–9980.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proc. North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

[9] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2011.

[10] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *Proc. the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, and et. al., "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[12] F. Pellegrini, "Distillating Knowledge about SCOTCH," in *Combinatorial Scientific Computing*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[13] J. Shulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," in *International Conference on Machine Learning*, 2017.

[14] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, "Learning generalizable device placement algorithms for distributed machine learning," in *Advances in Neural Information Processing Systems*, 2019.

[15] Y. Zhou, S. Roy, A. Abdolrashidi, D. L.-K. Wong, P. Ma, Q. Xu, A. Mirhoseini, and J. Laudon, "A single-shot generalized device placement for large dataflow graphs," *IEEE Micro*, vol. 40, no. 5, pp. 26–36, 2020.

[16] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," *arXiv preprint arXiv:1803.03288*, 2018.

[17] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," *arXiv preprint arXiv:1905.03960*, 2019.

[18] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[19] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[20] G. Karypis and V. Kumar, "Metis: Software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1998.

[21] B. Steiner, "Open source code of hierarchical planner in tensorflow," Jul. 2019. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/grappler

[22] C. Szegedy, V. Vanhoucke, S. Ioffe, and J. Shlens, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Computer Vision and Pattern Recognition (CVPR)*, 2016.

[23] 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, Łukasz 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," *CoRR*, vol. abs/1609.08144, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[24] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[25] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[26] "Tensorflow-slim library," Oct. 2019. [Online]. Available: https://github.com/tensorflow/models/tree/master/research/slim

[27] "Google's neural machine translation," Feb. 2019. [Online]. Available: https://github.com/tensorflow/nmt

[28] "Bert," Oct. 2019. [Online]. Available: https://github.com/google-research/bert