Accelerated Device Placement Optimization with Contrastive Learning

Hao Lan
University of Toronto
hao.lan@mail.utoronto.ca

Li Chen
University of Louisiana at Lafayette
li.chen@louisiana.edu

Baochun Li
University of Toronto
bli@ece.toronto.edu

ABSTRACT

With the ever-increasing size and complexity of deep neural network models, it is difficult to fit and train a complete copy of the model on a single computational device with limited capability. Therefore, large neural networks are usually trained on a mixture of devices, including multiple CPUs and GPUs, of which the computational speed and efficiency are drastically affected by how these models are partitioned and placed on the devices. In this paper, we propose *Mars*, a novel design to find efficient placements for large models. *Mars* leverages a self-supervised graph neural network pre-training framework to generate node representations for operations, which is able to capture the topological properties of the computational graph. Then, a sequence-to-sequence neural network is applied to split large models into small segments so that *Mars* can predict the placements sequentially. Novel optimizations have been applied in the placer design to achieve the best possible performance in terms of the time needed to complete training the agent for placing models with very large sizes. We deployed and evaluated *Mars* on benchmarks involving Inception-V3, GNMT, and BERT models. Extensive experimental results show that *Mars* can achieve up to 27.2% and 2.7% speedup of per-step training time than the state-of-the-art for GNMT and BERT models, respectively. We also show that with self-supervised graph neural network pre-training, our design achieves the fastest speed in discovering the optimal placement for Inception-V3.

CCS CONCEPTS

• **Computing methodologies** → **Sequential decision making**; *Distributed artificial intelligence*; • **Computer systems organization** → *Neural networks.*

KEYWORDS

device placement, contrastive learning, reinforcement learning, deep neural networks

ACM Reference Format:
Hao Lan, Li Chen, and Baochun Li. 2021. Accelerated Device Placement Optimization with Contrastive Learning. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472523

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472523

1 INTRODUCTION

It has been widely recognized that large neural network models lead to substantial gains in the quality of solving complex tasks. For example, Google's Bidirectional Encoder Representations from Transformers (BERT), as a breakthrough in natural language processing, contains up to 340 million parameters [6]. Training models with such sizes is time-consuming and resource-intensive, introducing significant practical challenges due to hardware constraints, such as memory limitations on GPU devices.

The need for neural network scaling and the limitations of computation resources, encourage machine learning practitioners to partition a large model across a heterogeneous mix of computational devices [2, 13, 28]. This is commonly referred to as "device placement" in the literature. One of the most challenging aspects when scaling up to large models is how their computation graphs should be partitioned and assigned to a heterogeneous set of computational devices, so that the limited resources on each device can be maximally utilized.

The decision of placing parts of the neural network models on devices is often made by human experts based on simple heuristics and intuitions, which are typically not flexible for a dynamic environment with many interferences. Deep reinforcement learning (DRL) has recently been proposed to provide effective device placements with full automation [7, 8, 21]. This type of approach follows the grouper-placer architecture. Manual grouping of model operations [7, 21] usually results in long searching time and is not flexible for finding the optimal placements, especially for giant neural networks with millions or even billions of parameters [26]. To enable automatic grouping, a hierarchical model [20] is proposed, which consists of a feed-forward neural network as a grouper and a placer instantiated as a sequence-to-sequence neural network with an attention layer. However, this hierarchical model experiences slow convergence of the learning process, as it involves two complex neural networks that are updated simultaneously.

In this paper, we propose our design, called *Mars*, a new DRL-based device placement mechanism that is fast and can be scalable to very large models. Rather than resorting to manual grouping, *Mars* adopts a graph neural network to encode the workload information and capture the topological properties of the computational graph. The encoded representations are then fed into a sequence-to-sequence neural network to generate placement. The design of *Mars* is based on a comprehensive array of empirical observations with regards to a number of design choices, including the selection of the graph encoder, the placer architecture, the training algorithm, and optimization. Specifically, we choose Deep Graph Infomax (DGI) [30] for node representations learning and leverage segment-level recurrent attention in the sequence-to-sequence placer. As

one of its salient advantages, *Mars* employs self-supervised pre-training of its graph encoder in its design, which achieves a further performance improvement.

We evaluate *Mars* by training it to produce device placement solutions for three benchmark machine learning models, *i.e.*, InceptionV3, GNMT, and BERT. The per-step execution times of our final placements are compared with the state-of-the-art, *i.e.*, Hierarchical Planner [20] and Generalized Device Placement [33]. Experimental results have demonstrated that the per-step execution time of the best placement discovered by *Mars* for GNMT and BERT is 2.7% and 27.2% shorter than the placements found by Hierarchical Planner. In addition, *Mars* achieves better training efficiency than the state-of-the-arts, it reduced its training time by an average of 13.2% via self-supervised pre-training.

2 PRELIMINARIES AND RELATED WORK

With the increasing size and complexity of deep neural networks, the computation and memory demands of deep learning have grown significantly. For example, a recently developed model, BERT [6], has millions of parameters and requires days of training with over a dozen Cloud TPUs. To meet such an ever-increasing demand for computing resources, it became mandatory to train deep neural networks in a cluster of heterogeneous computational devices, consisting of a mixture of CPUs and GPUs. As the model may not fit into the memory of a single computational device, *model parallelism* is widely used to partition a large model across multiple devices.

In this context, machine learning practitioners are given the flexibility to customize the mapping between devices and operations in their neural network models. Intuitively, different placement of operations may result in significantly different training times, depending on the utilization of resources and the overhead of communication. Thus, it becomes crucial to identify an optimal placement so that the total training time can be minimized. This problem, referred to as the *device placement* problem, is challenging to solve due to its combinatorial nature: it can be intuitively mapped to the graph partitioning problem, which has a variety of algorithms implemented in existing solvers (such as the Scotch optimizer [24], an open-source software library). However, they fail to achieve satisfactory results, as they require the construction of a cost model for a graph. Such a cost model is expensive to estimate and may not be accurate, especially for complex computational graphs in heterogeneous training environments.

To address the challenge, Mirhoseini *et al.* [21] proposed to use reinforcement learning to find the best placement. As shown in Fig. 1, given an initial placement, the neural network is trained for a few steps as a trial experiment to measure the training time, which is used as a reward signal for generating a better placement in the future with shorter training time. After a number of trial placements, an optimal placement with the shortest training time is eventually obtained.

Due to the slow convergence of reinforcement learning based on REINFORCE, a policy gradient method, the cost of finding the best placement is prohibitively high: in [21], it took 27 hours over 160 workers to find the best placement. With improved reinforcement learning algorithms based on proximal policy optimization (PPO) [27], Spotlight [8] and was able to reduce such a cost, while Post [7]

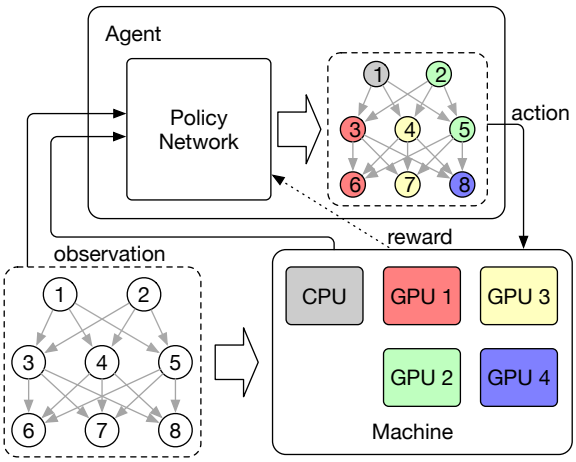
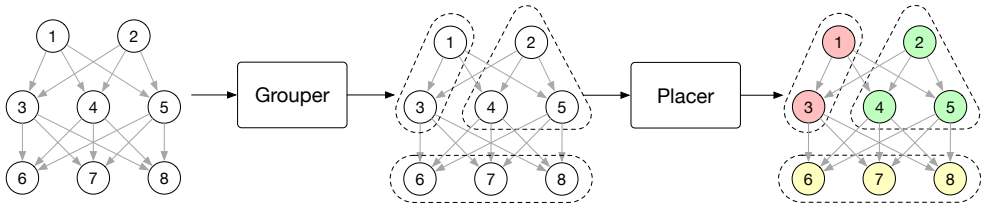


Figure 1: An illustration of using a reinforcement learning agent to place a neural network over multiple devices in a machine. The agent observes the computational graph and the states of the devices, and generates the mapping of the operations to devices.

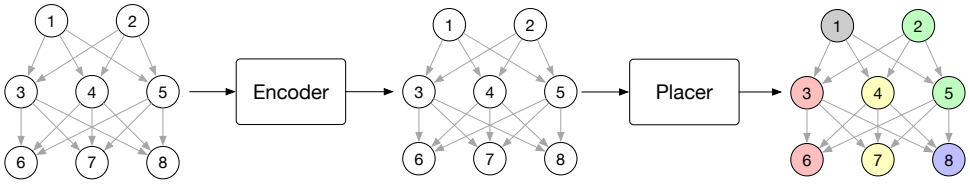
further combines PPO with the cross-entropy method to achieve even faster convergence and better placement for some neural network models. Besides the algorithm, EAGLE [19] proposed a more efficient grouper-placer-based method via an extensive exploration of different designs of the agent.

Essentially, an RL agent will need to compute a placement that assigns each operation to a computational device. With a large number of operations, the action space for the RL agent is huge, imposing a prohibitively heavy training workload. In practice, before being fed into the "placer" network in the RL agent, operations needs to be grouped first, which can be performed manually or with a "grouper," using heuristics or a feed-forward neural network [20]. As shown in Fig. 2, the grouper partitions operations into groups and merges the features of operations in the same group as group embeddings. The placer takes the group embeddings as input and predicts the placement for groups. This is referred to as the *grouper-placer structure*.

Recently, graph neural networks have shown great potential on inductive representation learning for graphs [9, 16, 30]. The state-of-the-arts [1, 22, 33] proposed to use a graph neural network to encode the operation features into trainable representations, which significantly improved the generalizability of the model. As shown in Fig. 2, Zhou *et al.* [33] replaced the grouper in the hierarchical model [20] with a graph neural network. The features of operations are encoded into embeddings (trainable representations) and then the placer can directly assign the device for each operation based on its embedding. Such an *encoder-placer structure* brings more flexibility and generality than traditional grouper-placer designs. However, it also introduces additional difficulty in training agent, such as larger action spaces and more sophisticated neural networks. In this paper, our primary focus is to address such difficulty by maximizing the training efficiency using a new and more efficient encoder-placer structure pre-trained with contrastive learning.



(a) Grouper-Placer model



(b) Encoder-Placer model

Figure 2: Two different architectures for device placement. a) The grouper-placer model reduces the action space by merging operations into groups; b) The encoder-placer encodes the features of the operations to capture the topological properties of the computational graph.

3 DESIGN

In *Mars*, we propose to take advantage of the salient properties of a graph encoder to maximize the training efficiency in the encoder-placer structure. As shown in Fig. 3, *Mars* employs a graph encoder pre-trained by contrastive learning and a lightweight segment-level sequence-to-sequence placer neural network [28]. The graph encoder is pre-trained in a self-supervised manner to initialize parameters to a good starting point, and the two components are trained jointly to generate the policy for operation placement. With our pre-trained encoder and lightweight placer, *Mars* achieves better training efficiency than the state-of-the-art: it discovers a better placement within a shorter period of training, to be elaborated in what follows.

3.1 Encoder Design

Essentially, the device placement problem is to identify an optimal placement across devices for a machine learning workload, which is described as a computational graph that consists of thousands of operation nodes with various attributes. Such a manually specified workload information is complex and difficult to be understood by a placer neural network directly, which requires a neural network (encoder) with delicate design to encode the information into machine-understandable and trainable representations. For extracting the underlying information from graph-structured data, graph neural networks (GNNs) have been employed in the state-of-the-art as the encoder, showing great promise. Typically, GNNs are built up by a few layers of graph convolutional networks (GCNs) [18]. They learn the representation of a node by aggregating features from its neighbor nodes. In this way, the learned node representations account for not only individual node features but also the graph structure.

Leveraging the favorable properties as aforementioned, *Mars* is designed with a GNN as an encoder to process the graph-structured information. In general, the encoder of *Mars* can be any GNNs as they share a similar representation learning strategy. For simplicity, we use a simple GNN, which consists of 3 GCN layers with Parametric ReLU (PRelu) activation function [12] as an example. A GCN layer takes two inputs, the adjacency matrix and node features of the workload graph, and generates the output as follows:

$$\mathrm{GCN}(\mathbf{X}, \mathbf{A}) = \sigma\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\boldsymbol{\Theta}\right) \tag{1}$$

where $\mathbf{X}$ denotes features or attributes of operations, $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops, representing the dependencies between operations in the computational graph, $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ is its diagonal degree matrix, σ is the PRelu activation function, and $\boldsymbol{\Theta}$ is the set of parameters of the GCN layers that we want to learn. Fig. 4 presents a clear illustration of feature aggregation performed by the GCN layers. As shown, the attributes of each operation node are gathered from the computational graph of the workload, including the operation type (*e.g.*, Conv2d, MatMul), input shape, and output shape (dimension). Since the operation type is not a scalar and the shape of operations' input and output may vary in a wide range of values, they cannot be fed into the encoder directly. Hence, we encode the operation types by one-hot encoding and normalize the shapes by the largest dimension size of all operations' input and output, and then feed them into the encoder as the node features. As shown in the figure, there is an edge (data flow arrow) from operation 1 to operation 3, which means operation 3 requires the output of operation 1 and it should be executed after operation 1 has completed. The GCN layers aggregate the features/attributes from neighbor operations along these edges, as observed for node 3 in the figure.

3.2 Pre-train Encoder with Contrastive Learning

To find better placements for machine learning workloads, *Mars* is designed with two submodels, the graph encoder and sequence-to-sequence placer, which are trained jointly in an end-to-end fashion.

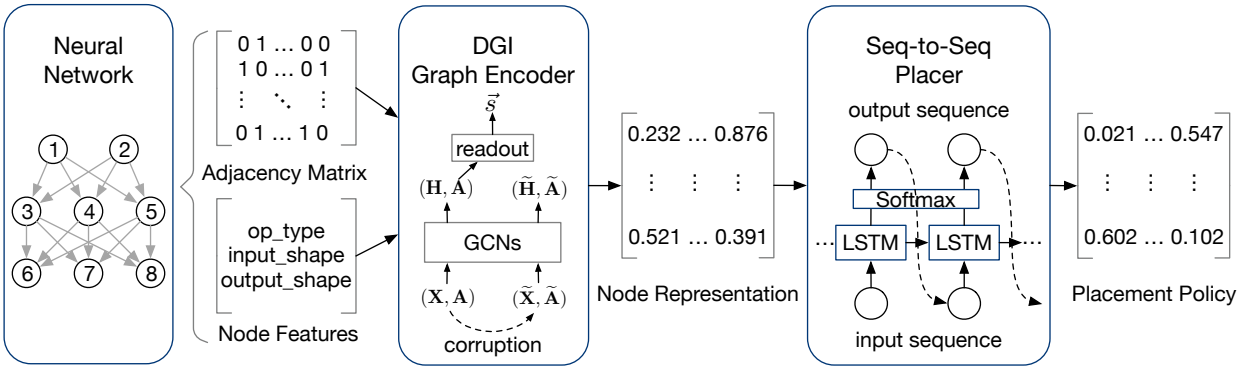


Figure 3: Illustration of Mars: a graph encoder consisting of a 3-layer graph convolutional network pre-trained via contrastive learning and a placer that is a sequence-to-sequence neural network for predicting the placement segment by segment.

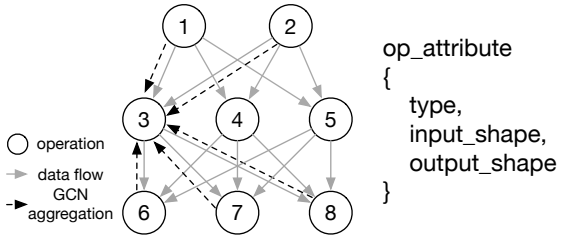


Figure 4: An illustration of feature aggregation performed by a graph convolutional layer in a computational graph.

However, it is challenging because neither of the two models is trivial to be trained. Large amounts of samples are required for model training, and evaluating these samples is expensive in the device placement problem. Fortunately, the pre-training of GNNs has been proposed recently to accelerate the training of downstream tasks [15]. Particularly, in GNN pre-training, a general graph task with a self-supervised learning objective will be designed, and the GNN can learn node representations from graph-structured data without labels. The learned representations are usually informative, able to comprehensively characterize the underlying semantics of a node within a graph. Upon the completion of pre-training, the trained GNN parameters can be used for other downstream tasks with minimal additional training costs.

Contrastive learning is one of widely used self-supervised pre-training methods for computer vision tasks [3, 4, 14, 29, 31] and graph tasks [25, 30, 32]. It learns representations by maximizing mutual information between differently augmented views of the same sample via a contrastive loss. The greatest advantage of contrastive learning is that it is self-supervised which means it does not require any labeled data. This is crucial especially for the tasks that require a decent amount of labels which is expensive or even impossible to get.

Inspired by such a promising type of technique, we propose to pre-train our graph encoder in a self-supervised learning approach proposed by Veličković *et al.* [30], which maximizes the mutual information between node representations and corresponding high-level summaries of the graph. More specifically, as illustrated by Fig. 3, we first generate a pair of samples in different augmented views, where the positive sample $(\mathbf{X}, \mathbf{A})$ is as the same as the original graph and the negative sample $(\widetilde{\mathbf{X}}, \widetilde{\mathbf{A}})$ is the graph augmented by a corruption function:

$$(\widetilde{\mathbf{X}}, \widetilde{\mathbf{A}}) \sim \mathcal{C}(\mathbf{X}, \mathbf{A}) \tag{2}$$

We use node permutation as the corruption function which shuffles the features between nodes. As illustrated in Fig. 5, the corrupted graph has exactly the same structure as the original graph except that the nodes are swapped. Next, we generate the representation $\vec{h}_i$ for each node i by aggregating features from the node and its neighbors via the graph encoder as Eq. (3):

$$\mathbf{H} = \mathbf{GCNs}(\mathbf{X}, \mathbf{A}) \tag{3}$$

The graph-level representation $\vec{s}$ is summarized by a readout function as Eq. (4), which simply averages representations of all the nodes in the graph:

$$\mathcal{R}(\mathbf{H}) = \sigma\left(\frac{1}{N}\sum_{i=1}^{N}\vec{h}_i\right) \tag{4}$$

Then, as shown in Eq. (5) below, we use a simple bilinear scoring function to score the mutual information between local information $\vec{h}_i$ and the global summary $\vec{s}$, and a logistic sigmoid nonlinearity to convert scores into probabilities:

$$\mathcal{D}\left(\vec{h}_i, \vec{s}\right) = \sigma\left(\vec{h}_i^T \mathbf{W} \vec{s}\right) \tag{5}$$

Finally, we minimize the Jensen-Shannon divergence between the probabilities of $(\vec{h}_i, \vec{s})$ being a positive example and $(\vec{\tilde{h}}_i, \vec{s})$ being a negative example:

$$\mathcal{L} = \frac{1}{N+N}\left(\sum_{i=1}^{N}\mathbb{E}_{(\mathbf{X},\mathbf{A})}\left[\log \mathcal{D}\left(\vec{h}_i, \vec{s}\right)\right] + \sum_{j=1}^{N}\mathbb{E}_{(\mathbf{X},\mathbf{A})}\left[\log \mathcal{D}\left(\vec{\tilde{h}}_j, \vec{s}\right)\right]\right) \tag{6}$$

By updating the parameters with gradient descent, informative node representations will be gradually learned throughout our self-supervised pre-training. After a few hundred steps, the pre-trained

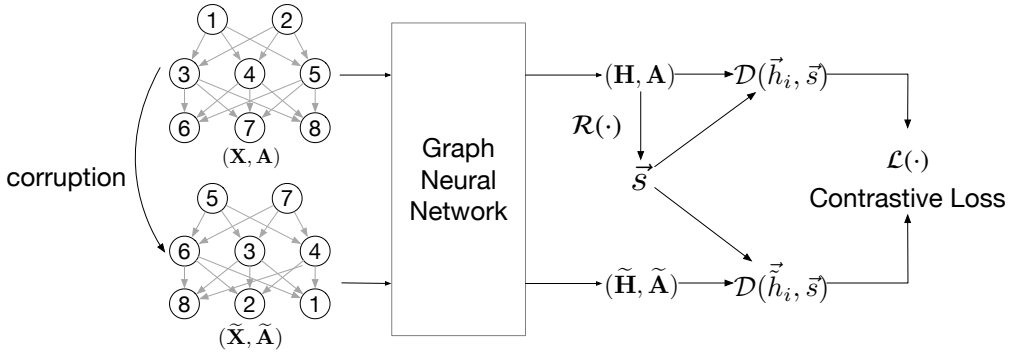


Figure 5: An illustration of pre-training graph neural networks with contrastive learning. Node permutation is used as the data augmentation method to generate negative samples. A global summary of the graph is generated by averaging representations of all nodes. Contrastive loss is calculated to maximize the mutual information between local representations and the global summary.

graph encoder can be directly used for our downstream task, device placement optimization.

Notice that, there are many choices of the corruption function, readout function and loss function in contrastive learning. In *Mars*, we simply follow the design used in Deep Graph Infomax (DGI) [30] for proof-of-concept. In general, other contrastive learning methods should also work for *Mars*.

3.3 Placer Design

After generating the node representations with the graph encoder, *Mars* uses a placer to process the representations further and learns how to place the operations optimally. When designing the placer model in *Mars*, we consider two popular neural network models in natural language processing (NLP): the sequence-to-sequence model [28] and Transformer-XL [5]. Both of them have been attempted in the literature for designing the placer, achieving satisfactory results [20, 21, 33]. Taking the node representation of operations as input, these placers generate placements in different ways. The sequence-to-sequence placer encodes all operations and outputs the selected device for each operation one by one. As the number of operations increases, it becomes less likely for the sequence-to-sequence placer to encode all of them at once efficiently. Zhou *et al.* [33] use a Transformer-XL based placer to encode and output the device of operations segment by segment, therefore it avoids processing the long sequence in one shot. However, we found that the Transformer-XL based placer is a little bit "heavy". Even for the simplest workload, InceptionV3, it takes a few thousands of training steps to converge.

Inspired by the sequence-to-sequence and Transformer-XL based placer, we propose to deploy a new placer model, a segment-level sequence-to-sequence model. It keeps the simplicity of the sequence-to-sequence model, which only has a bidirectional long short-term memory (LSTM) layer with an attention layer, and adopts the segment-level sequence processing that divides operations into small segments and places them at the segment level. As shown in Fig. 6, s is the segment size, $\{\vec{h}_1, \vec{h}_2 \dots \vec{h}_n\}$ is the node representation of operations and $\{p_1, p_2 \dots p_n\}$ is the placement where p_i is the device assigned for operation i. After encoding a segment of operations $\{\vec{h}_1, \vec{h}_2 \dots \vec{h}_s\}$ via a bidirectional LSTM, we directly predict (decode) the placement for this segment $\{p_1, p_2 \dots p_s\}$ via a unidirectional LSTM, such that the placer will focus on the segment currently under decoding. When the placer moves on to the next segment, it will take the encoded hidden states of the previous segment as the initial hidden state. This enables the placer to recall previous decisions when predicting the placement of the next segment. By doing so, the placer will continuously encode and predict the device of operations segment by segment until all operations are placed. The segment size s for sequence dividing is a hyperparameter, and we set it to 128 in *Mars*.

Table 1: Per-step training time (in seconds) of placements found by the agent with a trained graph encoder and different placers.

Models	Seq2seq	Trf-XL	Seq2seq (segment)
Inception-V3	0.100	**0.067**	**0.067**
GNMT-4	2.040	1.449	**1.440**
BERT	12.529	11.363	**9.821**

We evaluate these three placer designs experimentally with the same set of workload benchmarks: Inception-V3, GNMT, and BERT[1]. To eliminate the influence of the encoder, we train these three placers with fixed operation representations generated by the trained graph encoder, and evaluate the per-step runtime of best placements found by different placers. From the experimental results shown in Table 1, unsurprisingly, the sequence-to-sequence placer failed to find the best placement in all the benchmarks. The reason is that the sequence-to-sequence neural network is not good at processing long sequences, and it performs worse as the sequence becomes longer. Interestingly, we observed that the segment-level sequence-to-sequence placer outperformed the Transformer-XL placer in BERT, the largest of the three benchmarks, while achieving almost identical performance in the other two benchmarks. As BERT is a large and complex model, it is challenging for the placer

[1]The experimental setup will be described in the Section 4.

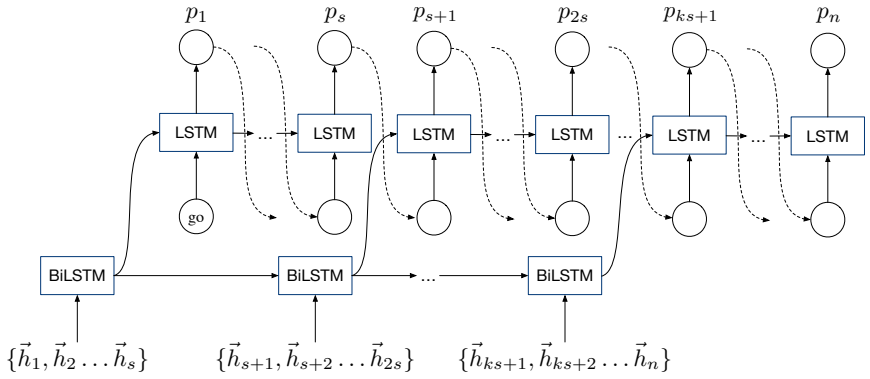


Figure 6: An illustration of the segment-level sequence-to-sequence placer. The operation sequence is split into multiple segments. A bidirectional LSTM is used to encode the segment of sequence, and a unidirectional LSTM is used to decode the segment. The encoded hidden state of previous segment is used as the initial state of encoding new segment.

to discover optimal placement with acceptable training overhead. The simplicity of the segment-level sequence-to-sequence placer enables it to converge better than the Transformer-XL placer, generating a placement with a shorter per-step runtime for BERT, as clearly shown in the table.

We also consider using a two-layer multilayer perceptron (MLP) as the simplest placer. However, it easily overfits, gets stuck at a local optimum and can never find a good placement. Based on the empirical observations and insights, we incorporate the segment-level sequence-to-sequence placer in our design, which is likely a sweet spot between two-layer MLP and Transformer-XL to achieve the best performance.

3.4 Joint Training with Reinforcement Learning

After pre-training the graph encoder with contrastive learning, we randomly initialize the placer and jointly train the two components of *Mars* in an end-to-end fashion using deep reinforcement learning. Specifically, proximal policy optimization (PPO) [27] is employed to update the policy of agent, which performs comparably or better than state-of-the-art approaches while being much simpler to implement and tune. PPO samples actions (placements) from the output policy and measures the per-step time of placements in a real-world environment (a multi-GPU machine). As shown in Eq. (7), we use the negative square root of the per-step time of placements as the reward. To estimate the advantage of placements, we use the exponential moving average of rewards as a baseline and calculate the advantages by subtracting the baseline from rewards:

$$\begin{aligned} R_t &= -\sqrt{r_t} \\ B_t &= (1-\mu)R_t + \mu B_{t-1} \\ \hat{A}_t &= R_t - B_t \end{aligned} \tag{7}$$

where r_t is the per-step time of the placement sampled at training step t, and B_1 is equal to R_1 since there is no B_0. μ is a hyper-parameter of the exponential moving average. We set μ to 0.99 for all experiments.

During the training, the agent may generate some invalid and bad placements. The invalid placements usually exceed the memory constrain of devices (out-of-memory) and cannot be run, so we assign an extremely long per-step time for them as a strong negative signal, such as 100 seconds. The bad placements are able to run but take much longer time than other placements. In some extreme cases, evaluating a bad placement may take over 20 minutes. To avoid wasting time on evaluating these bad placements, we apply a simple rule to terminate the evaluation. For example, if a placement of BERT takes longer than 20 seconds to finish a step, we will stop the evaluation and mark it as a bad placement.

4 EXPERIMENTAL RESULTS

In this section, we evaluate *Mars* with three widely used deep neural network models, and compare our results to four baselines. We analyze the training process of all RL-based methods to show why *Mars* can find a better placement than the baselines. We further evaluate the improvement of training efficiency due to self-supervised pre-training and discuss the generalizability of *Mars*.

4.1 Benchmarks and Baselines

We choose three typical deep neural network models for image classification and natural language processing tasks as our benchmarks for evaluation:

1) *Inception-V3.* An image classification model designed by Google. This model is relatively small and can easily fit into a single GPU. It is one of the baseline benchmarks used in existing RL-based device placement approaches in the literature. We use Inception-V3 to evaluate the ability of an agent to find the best placement. The batch size is set as 1.

2) *Google's Neural Machine Translation (GNMT).* We use the 4 LSTM layers version with an attention layer, where each LSTM layer has 256 hidden units. The sequence length is limited to the range of 20 to 50. We increase the batch size from 128 to 256. This makes it more challenging for the device placement problem, since the model requires more than 12GB GPU memory during the training which cannot fit into a single GPU. All the other settings are left as the default.

3) *Bidirectional Encoder Representations from Transformers (BERT),* which has a large number of operations and a complex design. It has

many variations of different sizes, BERT-Large, BERT-Base, BERT-Small and so on. We use BERT-Base with a maximum sequence length of 384 and a batch size of 24, which requires about 24GB GPU memory. Under this setting, the model has to be split across multiple GPUs and the communication between GPUs becomes the bottleneck.

We compare the performance of *Mars* to four baselines, two with pre-defined placements and two using state-of-the-art RL-based methods:

1) *Human Expert.* We use the hand-crafted placements defined in Google's implementation for these models. For Inception-V3, we use the implementation in the TF-Slim library. For GNMT, we use Google's NMT implementation, where each GNMT layer is assigned to each device in a round-robin manner. BERT, also developed by Google, does not support multi-GPU training using model parallelism by default.

2) *GPU Only.* This baseline places all GPU compatible operations on a single GPU while running incompatible operations on CPUs. This placement is only valid for smaller models such as Inception-V3 in our benchmarks, and will trigger an Out-Of-Memory error with larger models.

3) *The grouper-placer structure [20].* With a hierarchical design, the grouper-placer structure uses two neural network models, which are jointly trained with reinforcement learning. The grouper is a two-layer MLP and the placer is a sequence-to-sequence model with an attention layer.

4) *The encoder-placer structure [33].* The recent encoder-placer structure uses GraphSAGE as the graph encoder to replace the grouper in the grouper-placer structure, and an advanced sequence-to-sequence model, Transformer-XL, is used as the placer.

4.2 Experimental Setup

Following the convention of existing work, we train *Mars* for device placement using the following settings:

Hardware and software. The reinforcement learning environment in our training is a single physical machine, which has 4 NVIDIA P100 Pascal GPUs (each has 12GB RAM), 2 Intel E5-2650 v4 Broadwell @ 2.2GHz CPUs, and 125GB memory. For the software environment, we use Python 3.6, CUDA 9.0 and TensorFlow 1.12 for running deep neural network models in our benchmarks. Different from the environment, the agent of *Mars* is implemented using PyTorch 1.4.0. After the agent sampled the placement from its policy network, we will send the placement to the RL environment and measure the per-step runtime on the physical machine.

Performance evaluation. We evaluate the performance of device placement using the per-step runtime in our benchmark workloads. During the training of our agents, we only run the benchmark workload for 15 steps in each placement. When changing the placement of a benchmark workload, it needs to be re-initialized and warmed up for a few steps, which usually takes a longer time, so we discard the first 5 steps and average the per-step time of the last 10 steps. After the placement is finalized, we train the benchmark workload with the best placement for 1,000 steps and record the average per-step runtime as the final result.

Architecture of the agent. As mentioned in Section 3, *Mars* employs an encoder-placer structure. We explored a set of encoder designs and found that three-layers of GCNs with 256 hidden units performed the best. Our placer design in *Mars* is a segment-level 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 hidden LSTM layers have a size of 512. The attention mechanism we used is a context-based input attention mechanism [2]. The length of the segment is set to 128 in our experiments.

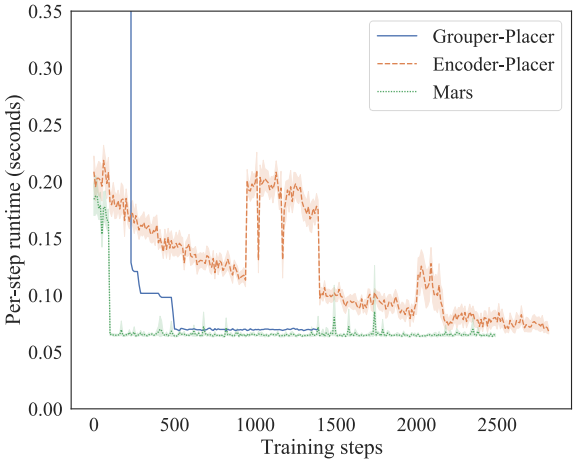
Training approach. Before training *Mars* with reinforcement learning, we pre-train the graph encoder with contrastive learning for 1000 iterations and save the parameters corresponding to the lowest loss. Then, we use proximal policy optimization to train the encoder and placer of *Mars* jointly. During the RL training, we sampled ten placements from each policy generated by the agent. For every 20 sampled placements, we shuffled them into four mini-batches and performed updates on each of the individual mini-batches. After repeating this for three epochs, the agent will generate new policies with updated parameters. The clip ratio, ϵ, is set to 0.2, and the coefficient of entropy is set to 0.001. We use Adam optimizer with a learning rate of 0.0003 and gradient clipping with a 1.0 norm.

4.3 Results and Analysis

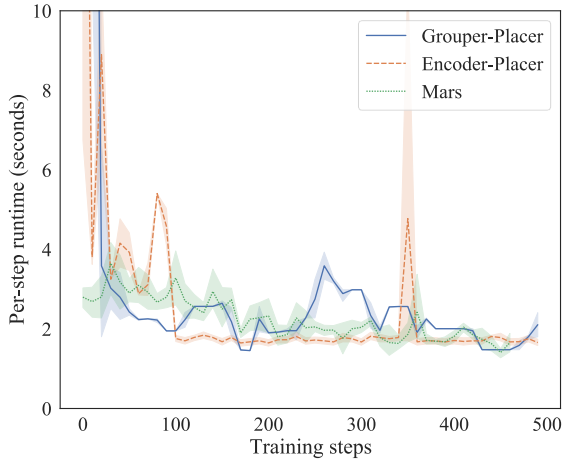
The analysis of training process. We first compare the agent training time for the three RL-based methods in Fig. 8. Then, we analyze the training process of all approaches with the Inception-V3 and GNMT-4 benchmark. Fig. 7 shows the per-step runtime of placements found during the training process. We averaged the per-step time of placements sampled from the same policy, and discarded all the invalid placements and some extremely poor placements with per-step runtime longer than 20 seconds.

As shown in Fig. 8, with self-supervised pretraining, *Mars* takes much less time than the other two alternatives for placing Inception-V3. As illustrated in Fig. 7a, *Mars* found the optimal placement for Inception-V3 within 100 steps, while the grouper-placer structure experienced a large number of invalid placements at the beginning of training. After learning with 200 placements, it started to generate valid placements and tried to further optimize the placement. Finally the grouper-placer structure found the optimal placement after 600 steps. While *Mars* converged with the fastest speed, the encoder-placer structure was much slower than either *Mars* or the grouper-placer structure, which took about 2500 steps to fully converge to the optimal placement for Inception-V3. In Fig. 8, we observed that all RL-based methods were able to find the optimal placement for GNMT-4 within 5 hours. Although the encoder-placer structure converged first, as Fig. 7b shows, it fell into a local optimum and failed to find a better placement until the end of training. Both the grouper-placer structure and *Mars* found a good placement at around the 200-th step. They kept exploring and finally identified the best placement at the 450-th step. For placing BERT, *Mars* trains the RL agent at the similar speed as the Grouper-Placer baseline. However, it finds a much better placement than the other two approaches.

From our results, we observed that with self-supervised pre-training, *Mars* always benefited from a better starting point than its rivals. It not only saves the training time of *Mars* but also helps



(a) Inception-V3



(b) GNMT-4

Figure 7: Per-step runtime of the placements found during optimizing the benchmark workloads Inception-V3 and GNMT-4 by different approaches.

Mars to find the best placement more easily. As a result, *Mars* outperformed the grouper-placer structure and the encoder-placer structure both in training efficiency and per-step runtime of the final placement.

The training efficiency improved by self-supervised pre-training with contrastive learning. To achieve better training efficiency, contrastive learning is employed in *Mars* to pre-train the graph encoder without interacting with a real environment which is slow and expensive. To evaluate its benefit, we first compare the per-step runtime of final placements found by the agent with and without self-supervised pre-training. As presented in Table 2, the agent in both setups can identify similar final placements, with self-supervised pre-training outperforming no pre-training.

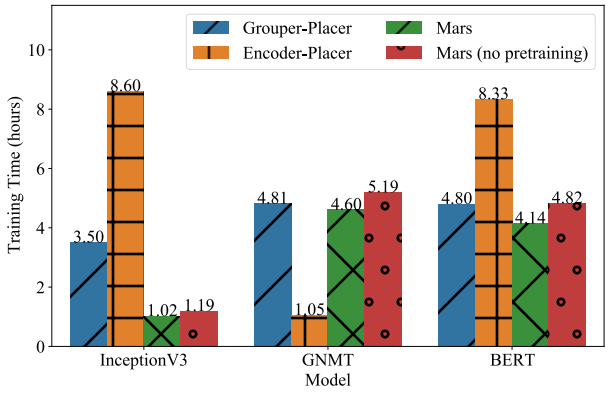


Figure 8: The training time (in hours) of the agent with different reinforcement learning based approaches.

Moreover, with respect to training efficiency, self-supervised pre-training shows a substantial amount of improvement and reduces the training time by 13.2% on average, compare to no pre-training, as shown in Fig. 8.

The reason is that contrastive learning can learn an informative representation for the operations of benchmark workloads before the agent start training with reinforcement learning. With learned representations, the agent can explore the possible placement from a better start point than a random initialization. For example, in Fig. 7b, the per-step runtime of placements found by *Mars* are all shorter than 4 seconds, even at the beginning of the training. While both the other two RL-based methods generated many bad placements longer than 10 seconds at first 50 steps. Evaluating a bad placement would take 4 times of time compare to a good placement. Thus, a better start point can save a substantial amount of the training time in reinforcement learning. Furthermore, since these representations encoded both the structure the computational graph and the features of the operations deeply, the agent was able to predict the placement with a better understanding. Hence, with self-supervised pre-training, the agent can find a better placement with less training steps.

The quality of final placements. As shown in Table 2, in the comparison between *Mars* and the two RL-based alternatives, *Mars* matches the performance (if it is the best) in Inception-V3, and outperforms both alternatives in GNMT and BERT.

With Inception-V3, all three alternatives are able to find the best placement, which is to place most of the operations on the same GPU device. Such a placement achieves similar performance compare to the GPU only baseline. The reason is that the computation power of a single GPU device is almost sufficient to meet the computation demand of this benchmark. The communication overhead incurred by using more devices outweighs the benefits of more computation power. The RL agents learned this insight and tried to reduce the communication overhead as much as possible. Meanwhile, the RL agent also noticed that some operations are faster when running on CPU. As a result, the best placement found by RL-based methods is slightly better than the baseline GPU only. This

Table 2: Per-step runtime (in seconds) of the best placements found by different approaches.

Models	Human Experts	GPU Only	Grouper-Placer	Encoder-Placer	Mars	Mars (no pre-training)
Inception-V3	0.071	0.071	**0.067**	**0.067**	**0.067**	**0.067**
GNMT-4	1.661	OOM	1.418	1.437	**1.379**	1.396
BERT	OOM	OOM	12.661	11.737	**9.214**	11.363

benchmark is also used in [7, 20, 21], and *Mars* achieved identical results reflecting the performance of the best possible placement.

With GNMT, *Mars* outperformed both its rivals in this benchmark workload, and all three alternatives found better placements than the human expert, which reduced the per-step runtime by 14.6%, 13.4% and 17.0%. With respect to BERT, our most challenging benchmark workload, the model is too large to fit into a single GPU. We discovered that all the alternatives were able to find a valid placement, but *Mars* again outperformed both state-of-the-art alternatives by an even wider margin.

We observed that the improvements reported in our experiments are not as significant as those reported in existing works. This is most likely due to the differences in hardware and software used in our experiments. The much more recent releases we used for both our hardware and software may have dramatically reduced the computation time in our benchmark workloads even without using reinforcement learning agents, and the potential for improvement in parallelizing these benchmark workloads across multiple devices will therefore become less substantial.

Table 3: Per-step training time (in seconds) of placements found by the agent pre-trained with similar and different types of DNNs.

Unseen workloads	Direct training	Generalized from similar type	Generalized from different type
Inception-V3	**0.067**	**0.067**	**0.067**
GNMT-4	**1.379**	1.422	1.472
BERT	**9.214**	10.127	12.426

The generalizability of *Mars*. As training the reinforcement learning agent for device placement problem is expensive, the state-of-the-arts generalize the agent by training it over a set of workloads, and using the learned policy to decide placement for unseen workloads. To evaluate the generalizability of *Mars*, we setup two scenarios in our experiments: the unseen workload has similar and different types with the training workload, respectively. Instead of predicting the placement for unseen workload directly, we fine-tune the policy for 100 steps to improve the quality of generated placement.

In Table 3, we compare the per-step runtime of the final placements found by directly training and generalizing policy learned from similar and different types of workload to unseen workload. For generalizing to the similar type of workloads, we choose VGG16, sequence-to-sequence and transformer as training workload respectively; GNMT-4, Inception-V3 and VGG16 are selected for generalizing to a different type of workload. For a fair comparison, we use the same number of total training steps in the evaluation. We first train *Mars* for optimizing the training workload until the agent cannot find better placement for 100 steps, and fine-tune *Mars* for the unseen workload for 100 steps. So the total training cost of generalization of *Mars* would be the training cost for its training workload plus 100 fine-tuning steps. Then, we use the same number of steps to directly train *Mars* for optimizing the test workload from scratch.

The result shows that *Mars* can generalize across different workloads, and the generalizability is intuitively better when the training workload and unseen workload are of a similar type. However, in terms of the per-step runtime of the final placement, the generalized agent is worse than direct training, especially for BERT. For Inception-V3, both the agent generalized from similar and different types can find the optimal placement for the workload. This is because *Mars* converges within 100 steps for Inception-V3 when directly trained from scratch, as shown in Fig. 7a. Therefore, 100 fine-tuning steps are sufficient for the generalized agent to converge. For GNMT-4 and BERT, the agent requires more training steps to find the best placement, and a better starting point can significantly contribute to faster convergence. If we allow the agent to fine-tune the policy with more training steps, it finally can achieve similar performance as direct training. However, the overhead of fine-tuning is close to training the agent from scratch.

Both generalization and self-supervised pre-training are trying to initialize the agent at a good starting point, while self-supervised pre-training can provide a decent starting point with much less overhead. To conclude, *Mars* with self-supervised pre-training is also a kind of "generalization", where the agent is "generalized" from maximizing mutual information of node representations to optimizing device placement.

5 MORE RELATED WORKS

While Mars is focused on optimizing the placement of neural networks over multiple devices of a single physical machine, some relate works try to speed up the training of the neural networks in a distributed scenario. TicTac [11] accelerates a distributed deep learning system by communication scheduling. It consists of two heuristics for efficient scheduling and improves iteration throughput by 20%. Priority-based Parameter Propagation (P3) [17] also improves 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 [10] combines traditional data parallelism with model parallelism enhanced with pipelining. It automatically partitions a neural network and pipelines them across multiple machines. HetPipe [23] further improves this pipeline parallelism by considering the heterogeneity of devices when partitioning the workloads. It groups a mixture of devices into a virtual worker such that each worker has similar computational resources, and then partition and pipeline the neural network across multiple virtual workers. Different from

these works, *Mars* is focus on single machine scenario which does not involve the network. Further, *Mars* only uses model parallelism which means there are no changes made to the original neural networks' architecture and also no staleness introduced.

6 CONCLUDING REMARKS

As neural network models keep growing in size and complexity, training these large models requires multiple computation devices. It becomes an important yet challenging research problem to find an optimal way to partition a large model with a huge number of operations and to place them on participating computation devices so that the training workload completes in the shortest period of time. In this paper, we have presented *Mars*, a unified reinforcement learning framework using a pre-trained graph encoder to encode features of operations in a neural network model into node representations, as well as a segment-level sequence-to-sequence placer to learn the best device placement using these representations. In particular, our design is based on a comprehensive analysis and experimental evaluation of a rich set of design choices for the model architecture and training approaches. We have implemented and evaluated *Mars* with three benchmark workloads: Inception-V3, GNMT, and BERT. Compared with existing work, *Mars* has demonstrated its superiority in discovering better placements for large GNMT and BERT models. For a relatively small workload (*e.g.*, Inception-V3), *Mars* is able to find the optimal placement in the smallest amount of time. We also analyzed the improvement of efficiency due to self-supervised pre-training, and the generalizability of *Mars*.

ACKNOWLEDGMENTS

The co-authors would like to acknowledge the gracious research support from Huawei Technologies Canada Co., Ltd., as well as the grant from the Louisiana Board of Regents under Contract Numbers LEQSF(2019-22)-RD-A-21 and LEQSF(2021-22)-RD-D-07, and National Science Foundation under Award Number OIA-2019511.

REFERENCES

[1] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[2] D. Bahdanau, C. Kyunghyun, and Y. Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proc. International Conference on Learning Representations (ICLR)*.

[3] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. In *Proc. International Conference on Machine Learning (ICML)*.

[4] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E. Hinton. 2020. Big Self-Supervised Models are Strong Semi-Supervised Learners. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[5] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V Le, and R. Salakhutdinov. 2019. Transformer-XL: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[7] Y. Gao, L. Chen, and B. Li. 2018. Post: Device placement with cross-entropy minimization and proximal policy optimization. In *Proc. Advances in Neural Information Processing Systems (NeurIPS*.

[8] Y. Gao, L. Chen, and B. Li. 2018. Spotlight: Optimizing Device Placement for Training Deep Neural Networks. In *Proc. International Conference on Machine Learning (ICML)*.

[9] W. Hamilton, Z. Ying, and J. Leskovec. 2017. Inductive representation learning on large graphs. In *Proc. Advances in Neural Information Processing Systems (NeurIPS*.

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

[11] S. H. Hashemi, S. A. Jyothi, and R. H Campbell. 2018. TicTac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288* (2018).

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proc. IEEE International Conference on Computer Vision (ICCV)*.

[13] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. IEEE Computer Vision and Pattern Recognition (CVPR)*.

[14] R. Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Philip Bachman, Adam Trischler, and Yoshua Bengio. 2019. Learning deep representations by mutual information estimation and maximization. In *Proc. International Conference on Learning Representations ICLR*.

[15] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay S. Pande, and Jure Leskovec. 2020. Strategies for Pre-training Graph Neural Networks. In *Proc. International Conference on Learning Representations (ICLR)*.

[16] Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, and Yizhou Sun. 2020. GPT-GNN: Generative Pre-Training of Graph Neural Networks. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

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

[18] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proc. International Conference on Learning Representations (ICLR)*.

[19] Hao Lan, Li Chen, and Baochun Li. 2021. EAGLE: Expedited Device Placement with Automatic Grouping for Large Models. In *Proc. IEEE International Parallel & Distributed Processing Symposium (IPDPS)*.

[20] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Quoc V. Le, and J. Dean. 2018. A Hierarchical Model for Device Placement. In *Proc. International Conference on Learning Representations (ICLR)*.

[21] A. Mirhoseini, H. Pham, Q. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proc. International Conference on Machine Learning (ICML)*.

[22] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2019. REGAL: Transfer Learning For Fast Optimization of Computation Graphs. *arXiv preprint arXiv:1905.02494* (2019).

[23] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *Proc. the 2020 USENIX Annual Technical Conference (ATC)*.

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

[25] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. GCC: Graph Contrastive Coding for Graph Neural Network Pre-Training. In *Proc .ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*.

[26] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Proc. International Conference on Learning Representations (ICLR)*.

[27] J. Shulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2017. Proximal Policy Optimization Algorithms. https://arxiv.org/pdf/1707.06347. In *Proc. International Conference on Machine Learning (ICML)*.

[28] I. Sutskever, O. Vinyals, and Q. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS*.

[29] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation Learning with Contrastive Predictive Coding. *CoRR* abs/1807.03748 (2018). arXiv:1807.03748 http://arxiv.org/abs/1807.03748

[30] P. Veličković, W. Fedus, W. L Hamilton, P. Liò, Y. Bengio, and R D. Hjelm. 2018. Deep graph infomax. *arXiv preprint arXiv:1809.10341* (2018).

[31] Qizhe Xie, Zihang Dai, Eduard H. Hovy, Thang Luong, and Quoc Le. 2020. Unsupervised Data Augmentation for Consistency Training. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[32] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. 2020. Graph Contrastive Learning with Augmentations. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[33] Y. Zhou, S. Roy, A.i Abdolrashidi, D. Wong, P. C Ma, Q. Xu, M. Zhong, H. Liu, A. Goldie, A. Mirhoseini, et al. 2019. GDP: Generalized Device Placement for Dataflow Graphs. *arXiv preprint arXiv:1910.01578* (2019).