Graph Representation Matters in Device Placement

Milko Mitropolitsky
KTH Royal Institute of Technology
Stockholm, Sweden
milkom@kth.se

Zainab Abbas
KTH Royal Institute of Technology
Stockholm, Sweden
zainabab@kth.se

Amir H. Payberah
KTH Royal Institute of Technology
Stockholm, Sweden
payberah@kth.se

Abstract

Modern Neural Network (NN) models require more data and parameters to perform ever more complicated tasks. One approach to train a massive NN is to distribute it across multiple devices. This approach raises a problem known as the *device placement* problem. Most of the state-of-the-art solutions that tackle this problem leverage *graph embedding* techniques. In this work, we assess the impact of different graph embedding techniques on the quality of device placement, measured by (i) the execution time of partitioned NN models, and (ii) the computation time of the graph embedding technique. In particular, we expand Placeto, a state-of-the-art device placement solution, and evaluate the impact of two graph embedding techniques, GraphSAGE and P-GNN, compared to the original Placeto graph embedding model, Placeto-GNN. In terms of the execution time improvement, we achieve an increase of 23.967% when using P-GNN compared to Placeto-GNN, while GraphSAGE produces 1.165% better results than Placeto-GNN. Regarding computation time, GraphSAGE has a gain of 11.569% compared to Placeto-GNN, whereas P-GNN is 6.95% slower than it.

CCS Concepts • **Computing methodologies** → **Neural networks**.

Keywords Distributed Deep Learning, Model Parallelization, Graph Embedding

ACM Reference Format:
Milko Mitropolitsky, Zainab Abbas, and Amir H. Payberah. 2020. Graph Representation Matters in Device Placement. In *Workshop on Distributed Infrastructures for Deep Learning (DIDL'20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429882.3430104

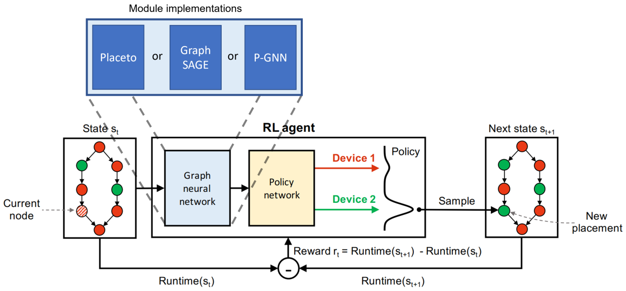

Figure 1. The Placeto device placement pipeline (the image is adapted from [1]).

1 Introduction

Nowadays, Neural Networks (NN) are increasingly being used for complex tasks. Such NNs usually need massive training datasets and a large number of model parameters to keep up with the performance requirements in terms of high accuracy. However, training big NN models with large datasets is computationally expensive and time-consuming. One approach to scaling the training process is to *parallelize* the training task by splitting it over multiple machines (*devices*) of a cluster.

In general, there are two common ways to implement parallelization in NNs, namely *data parallelization* and *model parallelization* [2]. In data parallelization, a NN model is replicated on multiple devices, and the training dataset is split among all the devices. Each device, then, trains its assigned NN in parallel to the other devices on a different part of the input training dataset. It is the current go-to strategy for spreading NNs across multiple devices [6, 12]. However, one assumption in this approach is that a NN is small enough, such that each device can load and process it. Thus, when a model requires more memory than that of a single device, the aforementioned technique falls short.

Model parallelization, on the other hand, is used mainly when a model is too large to fit on one device. In this approach, a NN and its computation is split across multiple devices, such that each device is assigned the computation task of a certain part of the NN. Unlike data parallelization, which is model agnostic, here a NN's architecture is critical to its partitioning strategy. *Device placement* is the study of how to split a NN and disseminate its nodes (operations) across different devices in order to minimize the training time, called the *execution time* [1, 8, 9, 16, 17].

The current trend in device placement solutions considers NNs as computation graphs and takes advantage of *graph*

DIDL'20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8206-9/20/12…$15.00
https://doi.org/10.1145/3429882.3430104

embedding techniques to make a representation of them before their partitioning [1, 10, 17]. Our main objective in this paper is to study the impact of different graph embedding techniques on device placement performance, evaluated by the *execution time* of partitioned NNs, and the *computation time* of the device placement process. To the best of our knowledge, such a comparison between graph embedding methods on device placement has not been made.

We base our work on Placeto [1], a state-of-the-art solution to the device placement problem, that leverages Reinforcement Learning (RL) to improve the placements for NNs iteratively. Placeto applies a graph embedding technique on the input NNs before feeding them to the RL method. Figure 1 shows the device placement pipeline in Placeto. Our contributions include the expansion of Placeto with additional state-of-the-art graph embedding models and the study of their impact on partitioned NNs' execution time and Placeto computation time. Concretely, we have implemented GraphSAGE [3] and P-GNN [15], in addition to the Placeto Graph Neural Network (Placeto-GNN), the default graph embedding model in Placeto [1]. Our updates in the Placeto pipeline are indicated in Figure 1.

Through our experiments, we show that P-GNN, a position-aware graph embedding, improves the execution time by 23.967% compared to the Placeto-GNN, and GraphSAGE produces 1.165% better results than Placeto-GNN. Regarding the computation time, GraphSAGE has a gain of 11.569% compared to Placeto-GNN, whereas P-GNN is 6.95% slower than Placeto-GNN.

2 Preliminaries

In this section, we briefly present some basic concepts from device placement and the graph embedding techniques we use in this work.

2.1 Device Placement

A NN architecture can be modeled as a directed graph $G = (V, E)$, where the operations of the NN are represented as vertices, and their connections as edges of the graph. Each vertex $v \in V$ denotes a node (operation) of the NN graph, and each edge $e_{u,v} \in E$ shows the connection between nodes u and v. Given $D = \{d_1, ..., d_m\}$ as the set of all the available devices, we want to divide V into n subsets, such that $\cup_{i=1}^{n} V_i = V$, and $n \leq m$. A *device placement policy* is defined as $\pi : V \longrightarrow D$ that assigns a vertex $v \in V$ to a device $d \in D$, i.e., $\pi(v) = d$. If $T(G, \pi)$ shows the execution time of G, partitioned by the placement policy π, then our goal is to find the *best placement policy* π^*, such that $T(G, \pi^*)$ is minimum.

Placeto [1] is a state-of-the-art device placement solution, based on which we conduct our study. It generalizes placement policies for previously unseen NN graphs. As an input, Placeto receives a NN G and an arbitrary device placement policy π. Then, it traverses all nodes $v \in V$, and for each node it performs the following steps: (i) first it uses Placeto's *graph embedding* model (Placeto-GNN) to create an embedding for G, given node v is selected, (ii) then, it uses an RL-based policy π to assign v to a device $d \in D$, and (iii) finally, it considers the execution time (training time) $T(G, \pi)$ using the new placement of node v and previous placement of the remaining nodes as the reward function to update the RL policy π. Thus, the algorithm iteratively proposes better placement policy π for consecutive nodes using the execution times as the reward function.

The required time to finish the steps above is called the *computation time*, i.e., the time to find the best placement policy π, and we denote it by $\mathcal{T}(G, \pi)$. To reduce the computation time, Placeto groups the nodes of G as proposed in [8] and creates graph G', such that $|V'| \ll |V|$, where $|.|$ denotes the number of nodes. Each node $v' \in V'$ consists of a subset of V. Therefore, Placeto improves computation time by traversing V', which is much smaller than V [1].

2.2 Graph Embedding

Graph embedding techniques are transformers that make low dimensional embeddings of nodes in large graphs [3]. Here, we briefly present the graph embeddings we use in our work: Placeto-GNN [1], GraphSAGE [3], and P-GNN [15]. The embedding principle in these techniques is to combine a node's features with an aggregation of that node's neighbourhood features using non-linear transformations.

Placeto-GNN. Placeto-GNN is the embedding technique presented in the Placeto paper [1]. To form the graph embedding of a node v, its features, including its total runtime, the output tensor size, and the current placement, are collected. Moreover, the information about the graph from the perspective of the node v is collected by passing messages to all v's parent nodes (nodes that can reach v via their outgoing connections), v's children nodes (nodes that can be reached by v via outgoing connections), and v's parallel nodes (nodes that cannot be reached by v). At the end, all these components are passed through a dense NN to form the final embedding.

GraphSAGE. GraphSAGE [3], initially, creates an embedding for each node v based on the node's features, including text attributes, node profile information, and node degrees. It then combines the embedding of each node with the aggregated features of its neighbourhood nodes. The neighbourhood of a node consists of layers of nodes up to distance K from that node. Given the neighbourhood of a node v, the feature aggregation starts with the nodes located K hops away from v. Since these nodes have no further neighbours within the current neighbourhood (i.e., v's neighbourhood), their embedding is only their local feature embedding. Nodes at the next level $(K - 1)$ form their embeddings by aggregating their neighbours' features from the previous level (K) and concatenating them with their features at the current level. Finally, this concatenation is passed through a fully connected layer with a non-linear function to create the next level’s input representation. The process is repeated K times until node v gets all the information needed to form its embedding.

P-GNN. One of the limitations of methods, such as Placeto-GNN and GraphSAGE, is that they miss the positioning and location information of the nodes while making the graph embedding, as they only capture the local neighbourhood of each node. This causes a problem in cases when two nodes with the same neighbourhood structure, are located in different parts of a graph, because they will end up in an identical place in the vector space of the graph embedding. P-GNN [15] tackles this issue by introducing sets of nodes within the graph, called *anchor sets*. To make a representation of a node, in addition to the neighbourhood aggregation, the information from the anchor sets is also taken into account. This additional information is weighed based on the distance of a node to the anchor set that enables capturing locality of the nodes in the graph. P-GNN first calculates the anchor sets in the input graph G. It, then, creates an embedding for each node v and its relation to each anchor set. The result is a vector, whose elements correspond to the aggregation of the relation of node v to each anchor set. This vector is passed through a fully connected layer to produce the final embedding of the node v.

3 Implementation

In this work, we aim to explore the impact of different graph embedding techniques on the current device placement methods. To this end, we conduct our study on Placeto [1], one of the latest solutions in device placement. As Figure 1 shows, the Placeto framework consists of two main components: the *graph neural network* and the *policy network*. The latter component is a multilayer fully connected network with a softmax layer that returns a probability distribution over the available devices for each node of the input NN. This component gets the graph representation as a vector generated by the former component, the graph embedding model, called Placeto-GNN. Here, we substitute the Placeto-GNN with GraphSAGE [3] and P-GNN [15], and study their impact on Placeto’s performance.

The graph representation comprises of the following features for each node: (i) the runtime of the node, (ii) its output size, (iii) the node placement, (iv) whether it is the current node, and (v) if the node has been visited before. The runtime of a node shows how much time it takes to compute the operation in that node. The output size of a node is the number of outgoing edges of the node. The node placement feature represents the device that the node is located currently, and the last two features are meta flags. In the Placeto implementation in [1], different operations are calculated on real hardware, and are used in the Placeto simulator to evaluate placement policies.

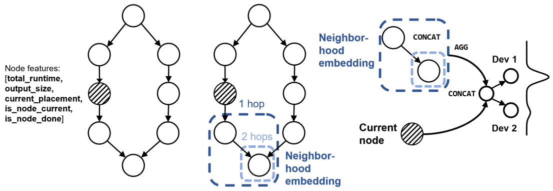

Figure 2. Example of GraphSAGE node embeddings. *Left:* Sample graph and node features. *Middle:* Neighbourhood embedding. *Right:* Aggregated neighbourhood embedding.

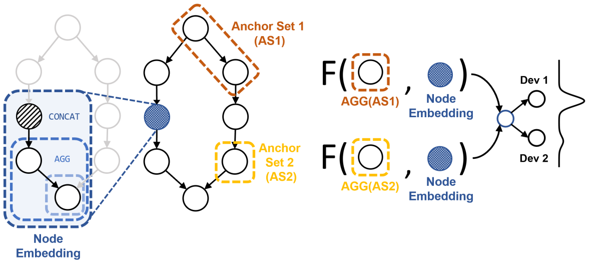

Figure 3. Creating node embeddings with the P-GNN implementation. *Left:* Create GraphSAGE embedding for the current node. Different sized random anchor sets (AS1 and AS2) are picked. *Right:* The relation between a node and each anchor sets is combined using function F.

3.1 GraphSAGE

Here, for each node, we first pass its aforementioned features through a simple multilayer fully connected NN to create the initial node embedding. We refer to the initial embedding of a node v as $\mathbf{h}_v^0$. Then, for each node v we choose a random sample of neighbouring nodes from its immediate neighbourhood. The sample size is a hyperparameter. We then proceed to build the subsequent embedding levels using the embeddings we have already created. The representation in each level k is computed as below:

$$\mathbf{h}_v^k = \texttt{fnn}(\texttt{concat}(\mathbf{h}_v^{k-1}, \texttt{agg}(\mathbf{h}_{N(v)}^{k-1}))), k \in [1, K],$$

where $\mathbf{h}_v^k$ is the embedding of node v at level k, and $\texttt{agg}(\mathbf{h}_{N(v)}^{k-1})$ is the aggregation of representation of v’s neighbours at previous level, $k-1$. We can apply any order invariant aggregation function, such as *mean* or *sum*, but we use *mean* in our implementation. The aggregation result is then concatenated with the node’s representation generated at the previous level. The output is given to fnn, a three layer fully connected NN, that creates the representation of this layer.

Figure 2 shows an example of how the embedding process works. Here, we assume the neighbourhood depth $K = 2$. For node v (the dashed one), we start from neighbourhood nodes two hops away and make their initial representations. Next, the embedding of the node one hop away from v is created by concatenating its representation with the aggregation of representation of the nodes two hops away from v. Since here there is only one node in each neighbourhood layer, the aggregation function returns its input. The result is then passed through the fnn (not shown in the figure), and its output is used in the next iteration. The same process is repeated for node v by concatenating its representation with the representation of the node one hope away (generated in the previous iteration), and giving it to fnn again.

3.2 P-GNN

We build P-GNN [15] using our GraphSAGE implementation and make the initial embedding in P-GNN as we do in GraphSAGE. To add locality to the embeddings, after the initial embedding of nodes, we choose the anchor sets S. The number of anchor sets is $c.log^2(n)$, where n is the number of the nodes, and c is a hyperparameter. The sets are of varying sizes, and their nodes are chosen randomly. The next step is to create an embedding for each node considering the anchor sets. To do so, we compute the aggregation of each anchor set and concatenate it with each node's embedding. The concatenation is then multiplied by the distance between the node and anchor sets.

More formally, the relation between a node v and an anchor set S_i is defined as below:

$$F(v, S_i) = \mathtt{dist}(v, S_i).\mathtt{concat}(\mathbf{h}_v, \mathbf{h}_{S_i}),$$

where $\mathbf{h}_v$ is the initial embedding of node v, and $\mathbf{h}_{S_i}$ is the embedding of anchor S_i, which is an aggregation of the embeddings of the nodes in S_i. We use *max* as the aggregation function in the implementation of $F(v, S_i)$. The function $\mathtt{dist}(v, S_i)$ is the distance between v and S_i. Given *max* as the aggregation function, this distance is the maximum distance between v and a node $u \in S_i$, where u is the furthest node in S_i to v. Conversely, the concatenation between $\mathbf{h}_v$ and $\mathbf{h}_{S_i}$ is equal to the concatenation of $\mathbf{h}_v$ and $\mathbf{h}_u$. When a node calculates its relation to each anchor set, those relations are aggregated into the ultimate node embedding using the *mean* aggregation. Figure 3 represents an example of how this process works. The initial embedding of a node v (the dashed one) is made using GraphSAGE. Then, two anchor sets S_1 and S_2 are selected, and $F(v, S_1)$ and $F(v, S_2)$ are computed as explained above.

4 Evaluation

In this section, we explain the experimental setup, and present the results of applying different graph embedding models on Placeto device placement performance.

4.1 Experimental Setup

We implement all the different models and conduct the experiments on the simulator provided by Placeto [1]. The settings of the simulator are identical in all the experiments, and we only change the graph embedding component, as shown in Figure 1. Here, we assume that each device has enough resource to store all the nodes of an input NN, thus for the *initial placement* we put all the nodes of each input NN on a single device. The purpose of this assumption is to show that even in situations that we can store the whole NN graph on a device, the model parallelization approach can improve its execution time. We also consider the placement using Placeto-GNN as the *baseline*. We use the following metrics for performance evaluation:

- *Execution time*: the training time of a NN after applying device placement policy based on a particular graph embedding technique.
- *Computation time*: the time to find the best device placement policy.
- *Execution-Computation (EC) ratio*: the relation between the execution time improvement and the computation time for a device placement policy.

Datasets

We use three datasets in the experiments as presented in the Placeto paper [1]. We refer to them as `cifar10`, `ptb`, and `nmt`. All three datasets are generated synthetically. `cifar10` and `ptb` are made by ENAS [4], an automated RL-based system that generates models by finding subgraphs of a bigger graph of operations. The `cifar10` dataset consists of convolutional NN graphs, whereas `ptb` contains recurrent NN graphs. `nmt` is generated by varying hyperparameters of the Neural Machine Translation (NMT) model [14]. To reduce the graph sizes, the nodes in all the graphs in all the datasets are grouped together, as proposed in [8]. After the operation grouping, the NN graphs in the `cifar10`, `ptb`, and `nmt` have on average 300, 500, and 190 nodes, respectively.

Training Environment

We conduct experiments on the described datasets using different numbers of devices: three, five, and eight devices. For each dataset, the placement policy assigns input NN nodes to these devices. That adds up to nine experiments in total (i.e., three datasets times three sets of devices). Each experiment runs the simulation over 51 randomly picked input graphs (17 graphs for each dataset). Each input graph passes through 20 episodes. We choose 20 episodes, as the improvements on average reach a plateau after about 10 to 15 episodes. All the details of our implementation are available on the following link[1].

4.2 Results

Execution Time. Here, we apply Placeto-GNN, GraphSAGE, and P-GNN on the three datasets and measure the execution time improvement compared to the initial placement (i.e., the input NN's nodes are on a single device). Table 1 shows the execution time improvement on `cifar10`, `nmt`, and `ptb` compared to the initial placement, respectively, on different numbers of devices. We see in Table 1 that in the experiment on `cifar10` over three devices, model parallelization in Placeto using Placeto-GNN provides a significant improvement to the initial placement with 24.585%. However, its improvement is less than the improvements provided by the GraphSAGE, 29.566%. P-GNN achieves the most prominent and most consistent improvement of 32.717%. Similarly,

[1] https://github.com/mmitropolitsky/device-placement

Table 1. The execution time improvements on `cifar10`, `nmt`, and `ptb` on different number of devices compared to the initial placement.

Graph Embedding	3 devices	5 devices	8 devices
	`cifar10`		
Placeto-GNN	24.585%	28.250%	34.339%
GraphSAGE	29.566%	30.567%	31.374%
P-GNN	32.717%	36.084%	37.114%
	`nmt`		
Placeto-GNN	28.107%	36.042%	38.813%
GraphSAGE	27.037%	34.177%	32.388%
P-GNN	30.232%	39.350%	42.399%
	`ptb`		
Placeto-GNN	8.981%	11.304%	15.831%
GraphSAGE	12.492%	14.658%	16.625%
P-GNN	17.930%	21.117%	23.536%

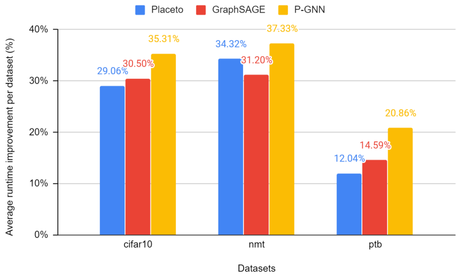

Figure 4. The average execution time improvement over different numbers of devices for the three datasets compared to theinitial placement.

we see the same pattern of improvement on five devices. The experiment on eight devices also confirms P-GNN provides the best results with 37.114% average execution time improvement. However, here Placeto-GNN performs better than GraphSAGE, and its difference with P-GNN is quite small, i.e., 2.775%. Table 1 also confirms that P-GNN has the most improvement on `ptb` and `nmt` datasets. Moreover, it shows that GraphSAGE works better than Placeto-GNN on the `ptb` dataset, but not on the `nmt` dataset.

Figure 4 shows the average execution time improvement over different numbers of devices for the three datasets compared to the initial placement. We observe that the larger the input graphs are, the smaller the average execution time improvement is achieved. For example, P-GNN reaches over 37% with `nmt` that contains the smallest graphs of around 190 nodes, and only around 20% with `ptb`, whose input graphs have 500 nodes in average. Regardless of graphs size, the locality added by P-GNN enables it to provide the best execution time improvements with any dataset.

Table 2. The average computation time (sec.) on `cifar10`, `nmt`, and `ptb` on different number of devices.

Graph Embedding	3 devices	5 devices	8 devices
	`cifar10`		
Placeto-GNN	110.730	112.030	108.118
GraphSAGE	89.116	91.930	93.443
P-GNN	101.170	105.721	103.420
	`nmt`		
Placeto-GNN	47.054	47.519	48.996
GraphSAGE	42.504	43.788	43.550
P-GNN	47.787	49.009	49.670
	`ptb`		
Placeto-GNN	335.376	327.175	321.450
GraphSAGE	288.443	298.736	298.210
P-GNN	362.761	368.460	371.815

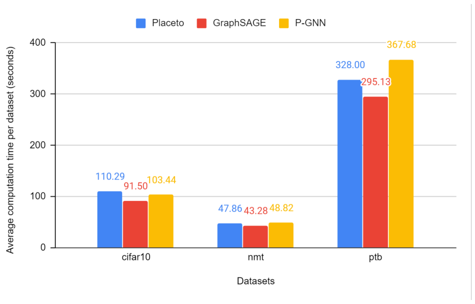

Figure 5. Average computation time per dataset in seconds.

Computation Time. Table 2 shows the average computation time of different graph embedding models on `cifar10`, `nmt`, and `ptb` datasets. In our experiments on the `cifar10` dataset with the different number of devices, the results consistently meet our expectations that the P-GNN requires more time to complete than GraphSAGE, due to more complex model. However, Placeto-GNN is the slowest of the three. Moreover, the experiments on the `nmt` and `ptb` datasets show that GraphSAGE needs the least computation time compared to the other models. Furthermore, in these two datasets, we observe that Placeto-GNN provides a better computation time than P-GNN.

Figure 5 represents the average computation time over the different number of devices for different datasets. Here, we can see that the computation time in all the graph embedding models is proportional to the size of the input graphs, i.e., the NNs in the `ptb` dataset require the most computation time, whereas the NNs in the `nmt` dataset require the least.

EC ratio. The EC ratio is a metric that gives a hint about the *usefulness* of a graph embedding model. It takes into account the execution time improvement compared to the initial placement, and relates it to the computation time. More formally, we define the EC of a NN G as $EC(G) = \frac{T(G,\pi)-T(G,\pi_0)}{\mathcal{T}(G,\pi)}$, where $T(G,\pi)$ and $T(G,\pi_0)$ are the execution time of training G partitioned by the placement policy π and the execution time of training G in the initial placement using π_0, respectively. $\mathcal{T}(G,\pi)$ is the computation time of training G partitioned by the placement policy π (Section 2).

As Table 3 shows, the average EC ratio of Placeto-GNN over all the datasets and the different number of devices is 0.009, GraphSAGE scores 0.01, and P-GNN has the highest ratio of 0.011. Despite having the slowest computation time, P-GNN still brings the highest EC gain due to its large execution time improvements. On the other hand, GraphSAGE achieves similar execution time improvements to Placeto-GNN. However, due to its comparatively fast speeds, it has a higher EC ratio than Placeto-GNN.

Table 3. Complete summary of all results for all tested parameters.

	Execution time improvements compared to the initial placement		**Computation times (seconds)**		**EC ratio**	
	Average	Improvement compared to the baseline	Average	Improvement compared to the baseline	Average	Improvement compared to the baseline
Placeto-GNN	25.139%	-	162.050	-	0.009	-
GraphSAGE	25.432%	**1.165%**	143.302	**11.569%**	0.01	**11%**
P-GNN	31.164%	**23.967%**	173.313	-6.95%	0.011	**22%**

Table 3 shows a summary of the graph embeddeing results that include the average of execution time improvement, the average of computation time, and the average of EC ratio over all the datasets and the different numbers of devices. Placeto-GNN provides an execution time improvement of 25.139% compared to the initial placement, and the computation time required is 162.050 seconds. GraphSAGE provides the execution time improvements very similar to Placeto-GNN, 25.432%, and the computation time is 143.302 seconds on average, which is 11.569% faster than the baseline. P-GNN has the most complex graph embedding architecture, and it achieves 31.164% execution time improvement compared to the initial placement, which is 23.967% more than Placeto-GNN. The computation time in P-GNN is 173.313 seconds on average, which is 6.95% slower than Placeto-GNN.

5 Related Work

J. Dean et al. present the initial idea of training large NNs in [2], where they introduce different parallelization issues: data-parallel and model-parallel training. R. Mayer et al. [7] present one of the earliest works on model parallelization and device placement, in which they propose several heuristics for this problem. One of the first AI-based device placement solutions was presented by A. Mirhoseini et al. [8], where they propose a method that learns how to optimize the device placement of TensorFlow graphs using RL. To achieve this, they model the RL policy as an encoder-decoder Recurrent Neural Networks (RNN). In [9], A. Mirhoseini et al. extend their previous work [8] by adding a grouper component to co-locate operations in groups using a softmax layer before feeding them to the decoder.

Spotlight [16] is another RL approach that uses Long Short-Term Memory (LSTM) [5] layer to produce device placements per group of devices. Spotlight uses Proximal Policy Optimization (PPO) [11] as the placement policy. Y. Zhou et al. propose GDP [17] to learn and generalize graph structures. This work, first, makes a representation of the input NNs using GraphSAGE [3], and then utilizes a Transformer network [13] for proposing device placement. Placeto [1] is the state-of-the-art solution in this domain that we covered in the previous sections, and thus we do not repeat it here. Although there are a few device placement solutions that use graph embedding, there is no study on the impact of this module on the execution time of the final placement, and this is the gap we tried to fill in this work.

6 Conclusions

In this work, we study the impact of different graph embedding models (i.e., Placeto-GNN, GraphSAGE, and P-GNN) on the device placement policies' performance. We measure the performance for the execution time of NNs partitioned across multiple devices, and the computation time of applying a graph embedding model on them. We conduct our study over Placeto, a state-of-the-art device placement solution. Through our experiments, we show that a position-aware graph embedding technique, such as P-GNN, can improve the overall device placement performance, i.e., the ratio of the execution time improvement to the computation time.

References

[1] R. Addanki et al. 2019. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv:1906.08879* (2019).

[2] J. Dean et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[3] W. Hamilton et al. 2017. Inductive representation learning on large graphs. (2017), 1024–1034.

[4] P. Hieu et al. 2018. Efficient neural architecture search via parameter sharing. *arXiv:1802.03268* (2018).

[5] S. Hochreiter et al. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[6] Y. Huang et al. 2018. Flexps: Flexible parallelism control in parameter server architecture. *VLDB Endowment* 11, 5 (2018), 566–579.

[7] R. Mayer et al. 2017. The tensorflow partitioning and scheduling problem: it's the critical path!. In *DIDL*. 1–6.

[8] A. Mirhoseini et al. 2017. Device placement optimization with reinforcement learning. *arXiv:1706.04972* (2017).

[9] A. Mirhoseini et al. 2018. A hierarchical model for device placement. (2018).

[10] A. Nazi et al. 2019. Gap: Generalizable approximate graph partitioning framework. *arXiv:1903.00614* (2019).

[11] J Schulman et al. 2017. Proximal policy optimization algorithms. *arXiv:1707.06347* (2017).

[12] A. Sergeev et al. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv:1802.05799* (2018).

[13] A Vaswani et al. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[14] Y. Wu et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv:1609.08144* (2016).

[15] J You et al. 2019. Position-aware graph neural networks. *arXiv:1906.04817* (2019).

[16] G. Yuanxiang et al. 2018. Spotlight: Optimizing device placement for training deep neural networks. In *ICML*. 1676–1684.

[17] Y. Zhou et al. 2019. GDP: Generalized Device Placement for Dataflow Graphs. *arXiv:1910.01578* (2019).