DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2020*

On the Impact of Graph Embedding on Device Placement

MILKO MITROPOLITSKY

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

On the Impact of Graph Embedding on Device Placement

MILKO MITROPOLITSKY

Master in Computer Science
ICT Innovation, Cloud Computing and Services
Date: August 17, 2020
Supervisor: Zainab Abbas
Examiner: Amir H. Payberah
School of Electrical Engineering and Computer Science
Host company: Logical Clocks
Swedish title: Påverkan av grafinbäddning av enhetsplacering

Abstract

Modern neural network (NN) models require more data and parameters in order to perform ever more complex tasks. When an NN model becomes too massive to fit on a single machine, it may need to be distributed across multiple machines. What policies should be used when distributing an NN model, and more concretely how different parts of the model should be disseminated across the various machines is called the *device placement* problem. Tackling the matter is the focus of this thesis.

Previous approaches have required the placement policies to be created manually by human experts. Since that method does not scale well, the current efforts to tackling the device placement problem focus on automating the process using reinforcement learning (RL). Most of the RL systems contain different kinds of graph embedding modules.

Our work tries to increase the knowledge about how to tackle the device placement problem by measuring and assessing the impact of graph embeddings on the quality of the device placement policies. We compare the different approaches in two main ways: *runtime improvement* and *computation time*. The former is a metric of how much faster is the new placement policy compared to a baseline. The latter describes how much time is required by the system to achieve that runtime improvement. In this thesis, we build on previous efforts in the device placement field in order to explore how different state-of-the-art graph embedding architectures affect device placement policies. The graph embedding architectures we compare are Placeto (used as a baseline), GraphSAGE and P-GNN.

In terms of runtime improvement, we achieve an increase of 23.967% when using P-GNN compared to Placeto, and 31.164% absolute improvement from the initial placement. GraphSAGE produces 1.165% better results than Placeto with the same setup. Regarding computation time, GraphSAGE has a gain of 11.560% compared to Placeto, whereas P-GNN is 6.950% slower than the baseline.

Given our results, we can confirm that graph embedding architecture can have a significant impact on device placement policies and their performance. More complex graph embedding architectures that capture more data about the graph and its topology provide better runtime improvements. However, that complexity may come at the cost of the computation time required to train the placement system.

Sammanfattning

Moderna neurala nätverk (NN) -modeller kräver mer data och parametrar för att utföra allt mer komplexa uppgifter. När en NN-modell blir för stor för att rymmas på en dator, kan den behöva distribueras över flera datorer. Vilka vilkor som ska användas vid distributionen av en NN-modell, och mer konkret hur olika delar av modellen ska spridas över olika datorer kallas enhetsplatsproblemet. Avhandlingen kommer att fokusera på detta problem. Tidigare tillvägagångssätt har krävt att placeringspolicyn skapas manuellt av människor med expertis i detta område. Eftersom den metoden inte går att skala upp fokuserar man på att hantera enhetsplaceringsproblemet genom att automatisera processen med reinforcement learning (RL). De flesta av RL-systemen innehåller olika typer av grafinbäddningsmoduler.

I arbetet försöker vi öka kunskapen om hur man hanterar problem med enhetsplacering genom att mäta och bedöma effekterna av grafinbäddningar på kvaliteten på villkoren för enhetsplacering. Vi jämför de olika metoderna på två sätt: runtime improvement and computation time. Den förstnämnda är ett värde för hur mycket snabbare den nya placeringspolicyn är i jämförelse med en baslinje. Det andra beskriver hur mycket tid som krävs av systemet för att uppnå den förbättrade runtime.Den här avhandlingen bygger på tidigare forskning inom området av enhetsplacering för att undersöka hur olika toppmoderna metoder till enhetsplaceringsprinciper. Grafinbäddningsarkitekturer som vi jämför i avhandlignen är Placeto (används som en baslinje), GraphSAGE och P-GNN.

Vi uppnår en förbättring av runtime med en ökning på 23.967% när vi använder P-GNN jämfört med Placeto och 31.164% ökning från baslinjen. GraphSAGE ger 1.165% bättre resultat än Placeto med samma installation. När det gäller beräkningstiden har GraphSAGE en förbättring på 11.560% jämfört med Placeto, medan P-GNN är 6.950% långsammare än baslinjen.

Med resultaten kan vi bekräfta att grafinbäddningsarkitektur kan ha en betydande inverkan på enhetsplaceringsprinciper och deras prestanda. Desto mer invecklad grafinbäddningsarkitekturer fångar mer data om grafen och dess topologi ger runtime improvment. Däremot blir kan komplexiteten kosta i computation time på grund av det tid som krävs för att utbilda placeringssystemet.

Абстракт

Модерните модели на невронни мрежи (МНМ) изискват все повече параметри, за да изпълняват все по комплексни задачи. Когато МНМ стане прекалено голям, за да се побере на едно устройство, той трябва да бъде разпределен върху множество машини. Какви политики трябва да се използват при дистрибутиране на МНМ и по-конкретно как различни части на модела трябва да се разположат върху различни машини се нарича проблемът за разположение на устройства. Този труд се фокусира върху справянето с този проблем.

Предишни подходи са налагали политиките за разпределение на модел да бъдат извършвани на ръка от експерти. Тъй като този подход не може да се скалира настоящите метод за справяне с проблема за разположение на устройства се фокусират върху автоматизацията на процеса чрез използване на обучение с утвърждение. Повечето такива системи имат разнообразни модули за вграждане на графове.

С този труд изследваме как може да бъде преодолян проблемът за разположение на устройства, като се измерва и оценява влиянието на влагания на графове върху качеството на политики за разположение на модел. Сравняваме различните подходи по две метрики: подобрение на времето за изпълнение и време за изчисление. Първата метрика показва какво подобрение носи новата политика в сравнение с началното ниво. Втората описва колко време е необходимо на системата да достигне дадено подобрение на времето за изпълнение. Ние изследваме как различни модерни архитектури за влагане на графове влияят на политиките за разпределение. Архитектурите за влагане на граф, които сравняваме са Placeto, GraphSAGE и P-GNN.

От гледна точка на подобряване на времето на изпълнение, ние постигаме 23.967% по-добро време използвайки P-GNN вместо Placeto, или 31.164% абсолютно подобрение спрямо първоначалното разположение. GraphSAGE постига 1.165% по-добри реултати от Placeto при същите условия. Относно времето за изчисляване, GraphSAGE е с 11.56% по-бърз от Placeto, докато P-GNN е с 6.95% по-бавен.

Постигнатите резултати потвърждават, че различните архитектури за влагане на графове оказват влияние върху качеството на политиките за разположение на устройства. По-комплексни архитектури, улавящи повече данни за структурата на графа и неговата топология показват по-добри резултати. Тази комплексност, обаче, идва на цената на увеличение на времето за изчисляване нужно на системата.

Acknowledgments

I am thankful to my examiner Amir H. Payberah for the guidance he provided while I was pursuing the goals of this thesis.

I am grateful to my supervisor Zainab Abbas for her diligence when reviewing and offering feedback to my work.

Thanks should also go to Sina Sheikholeslami for his dedication when providing me with the resources needed to achieve my results.

This thesis would not have been possible without the unwavering support of my family and friends.

Last but not least, I want to express my gratitude to Viktoriya Kutsarova for always pushing me forward.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural Networks (NNs) have become an inseparable part of our daily lives - our voice assistants, our photo tagging, face recognition, and many more. To achieve this, NN models have been growing in size and continue to do so. Machines have become sufficiently powerful to withstand resource demand. That ability of the machines is one of the reasons that Artificial Intelligence (AI), and in particular NN research have seen a resurgence in the early 2000s after the last AI Winter (Hsia (2018)).

However, there are several factors that cast a shadow over the bright picture of the current AI and NN landscape. On the one hand, the datasets that we have available are enormous, while still growing in size and scope. On the other hand, the NN models are growing in unison with the complexity of the problems that we want to solve. Furthermore, the very resource that the machines can provide and is expected to handle the load is not infinite. In fact, the exponential growth in computational power predicted by Moore's Law is underdelivering and current developments are falling of the curve (Economist (2016)).

From the perspective of an NN researcher, there are two main ways to deal with the boost in resource demand. The first option is to scale vertically. That implies adding or upgrading the hardware of a single machine, i.e., adding more RAM, or getting a more powerful CPU. It is a reasonable solution, but its limiting factor is the price of the newly added parts. The second option is to scale horizontally or to make a cluster of commodity machines work together and finish a task as if it was a single machine. This method does not have the price problem; however, other issues arise, such as synchronization, fail-safety, consistency of data, to name a few. Still, those problems are acceptable, and there are ways of dealing with them. That is why this method is used for

scaling larger systems even outside of the NN field, and we will focus our efforts in this direction.

There are two popular ways for scaling an NN model horizontally, namely *data parallelization* and *model parallelization*. In data parallelization, the model is replicated on each machine of the cluster, and each machine works on a different part of the input dataset. This approach is easy to implement, as it does not require specific setup for the NN model. On top of that, it does not care about what model it works with, as the model is replicated entirely. It is suitable for large datasets. Data parallelization has been studied (Recht et al. (2011), Dean et al. (2012), H. Li et al. (2015), H. Cui et al. (2016), Koliousis et al. (2019)) and is the current go-to strategy for spreading an NN model on multiple machines. However, when a model has memory or computational requirements that cannot be met by a single machine (or in other words the model does not fit on a single machine), this technique falls short.

Model parallelization, on the other hand, aims to solve the aforementioned issue. It is de facto a must use method if a model is too large. The approach is to split the computation of such a model across multiple machines, such that each machine is assigned the calculation of a certain part of the model. The machines execute the assigned work in parallel. Each datapoint of the dataset is propagated through all machines containing the model. Here, the luxury of model agnosticism and no additional work that data parallelization provides are gone. Synchronization and communication costs are increased. It is a trade-off, which is going to be faced more and more commonly. *Device placemenet* is the study of how to split an NN model and disseminate its operations across different machines, as to optimize runtime of the model and reduce the computation costs. This thesis focuses on the exploration of various device placement techniques.

1.1 Research Question

The issue with the device placement problem and model parallelism is that we have limited knowledge about how to split the model in such a way as to optimize the runtime of the distributed NN. Most of the previous methods are based on manual splitting by experts, which hinders the scalability of this method. On top of that, each model needs to be split separately, as there is no one-size-fits-all solution yet. Efforts have been made towards automating the process of device placement. However, since it is a rather new topic, many possibilities and options have not been explored.

Thus, the research question for this thesis is how to tackle the automated device placement for model parallelization?

1.2 Goals

The current state of the art solutions use *Reinforcement Learning* (RL) to tackle the device placement problem (Mirhoseini, Pham, et al. (2017), Mirhoseini, Goldie, et al. (2018), Addanki et al. (2019)). As part of their RL algorithms, many use different *Graph Embedding* approaches. However, since research in the device placement area has emerged quite recently with one of the first works being Mirhoseini, Pham, et al. (2017), many options for graph embedding architectures are yet to be explored. Thus, given the problem stated, the goal of this thesis is to analyze and compare the impact of different graph embedding architectures on the performance of an RL system for automatic device placement.

1.3 Thesis Contributions

The contributions of this thesis include:

- The expansion of an automated NN device placement framework based on RL with additional graph embedding architectures. We explore the impact different graph embedding architectures have on performance as part of automatic device placement system. Concretely, we have used the framework provided as reference code for Addanki et al. (2019), and have added the graph embedding methods presented in Hamilton, Ying, and Leskovec (2018) and You, Ying, and Leskovec (2019) on top of the one described in the original paper.
- We have conducted experiments based on the simulator provided in the aforementioned framework. The experiments measure the quality of the automatic device placement based on the improvement of simulated runtime. The experiments also explore the impact of the graph embedding methods on computation time needed by the simulator.

1.4 Methodology

Quantitative research methodology has been used for this thesis. Using the terminology introduced in Meredith et al. (1989), the methodology of this work

can be more precisely described as axiomatic research. Axiomatic research produces knowledge about the behaviour of different variables in the model, based on assumptions regarding the behaviour of others. In line with this, we achieve the goals of this thesis by utilizing a simulator used in Addanki et al. (2019).

Given that framework as a base, we compare and analyze the estimated runtime improvement of the simulations with the core graph embedding method of the framework replaced with different graph embedding architectures. We also evaluate the computation time needed to reach the runtime improvements. In order to fulfill the goals, we have conducted studies of the most recent literature in the fields of graph embeddings and neural network device placement.

1.5 Ethics and Sustainability

Device placement, and by extension, the usage of device placement methods with deep learning models can raise ethical concerns. For example, it is possible that certain device placement policies suggested by the algorithms in this work might introduce bias in the model's predictions. That bias might be in essence unethical in its inferences, if for example the model deals with classification and characterization of people. It is important to check if the model's predictions are ethically compromised after it has been automatically placed using the presented methods. Better understanding of device placement algorithms can lead to reducing such a possible impact.

With the increase in size of deep learning models, more resources are being used for training and inference. Resources, such as electrical power and materials used for hardware, need to be reduced. Exploring methods for optimizing the training and inference times, such as the ones presented in this thesis, can lead to reduction of the required resources, and thus contribute to a more sustainable development for machine learning and deep learning research.

Chapter 2

Background

Supporting the ever growing amounts of information, as well as artificial NN models that are increasingly more complex, requires a lot of resources. The computational power of a single machine can easily become insufficient to facilitate the training and inference processes of NNs. Increasing the power of a single machine can quickly become too expensive. That is why there is a need for methods that allow for multiple machines to combine their work and overcome the issue.

The purpose of this chapter is to give an overview of the background of the research question by presenting short summaries of the related topics, methods, and technologies. The section is divided into several parts. Each one introduces a concept that is required to understand this thesis.

First, an outline of artificial NNs is presented. It is comprised of a description of the technology and its impact.

Second, a brief introduction about the distribution and parallelism in the context of neural network models is presented. This part of the text describes the need for distribution of NN models, as well as two types of parallelism: data parallelism and model parallelism. An introduction to the Device Placement problem is presented. In that part we take a look at scalability issues in general and in paticular in NNs.

Third, a brief overview of RL is needed to gain a full view of the presented work, as the current state of the art regarding device placement relies on RL techniques to achieve better results.

Fourth, in order to understand the current state of the art solutions, the topic of graph embeddings is explained. Some recent developments in the field, such as Graph Neural Networks (GNN) are presented and discussed. The section covers some of state-of-the-arts in more details, as their approaches are pivotal

for this work.

Last, we lay out alternative approaches to tackling the device placement problem. Since this is the focal point of this thesis, a more thorough summary of the current work is presented.

2.1 Neural Networks

Understanding the device placement of NNs, requires certain knowledge about NNs themselves.

Artificial NNs (NNs for short) are not a new concept. In fact, the first work that proposed the idea was written by McCulloch and Pitts (1943). The concept is loosely related to biological NNs. When a (biological) neuron produces a signal, it travels via the axons to the synapses. The synapses are connected to the bodies of other neurons. If the synapses release a sufficient amount of signals, the latter neuron is activated and in turn sends its own signal forward.

In the paper, McCulloch and Pitts present an artificial structure that is able to mimic this behaviour. It is an artificial neuron that has multiple binary inputs and a single binary output (an on/off switch). A neuron is activated, if a certain number of its inputs are in turn activated. This allow for performing many logical tasks (e.g., AND, OR, etc). Combining such neurons allows for computing more complex tasks.

Rosenblatt (1957) built on top of the idea and proposed a new version of the artificial neuron, called *Linear Threshold Unit* (LTU) - see Figure 2.1. From now on we will use the term neuron to represent an LTU. The inputs are now numbers and each is associated with a weight. The LTU computes a sum of the inputs and applies a step function, the result of which is used as output. A single layer of LTUs is called a perceptron or simply a layer. Stacking multiple layers of perceptrons creates a multilayer perceptron, or a deep NN (Figure 2.2).

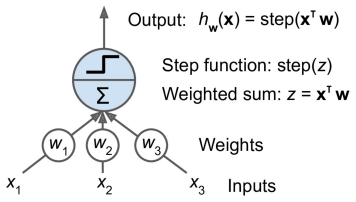


Figure 2.1: A *Linear Threshold Unit*. Source: (Géron 2017)

A deep NN by itself is not an effective tool to perform its intended tasks. In order to take advantage of a deep NN, it needs to be trained. When training an

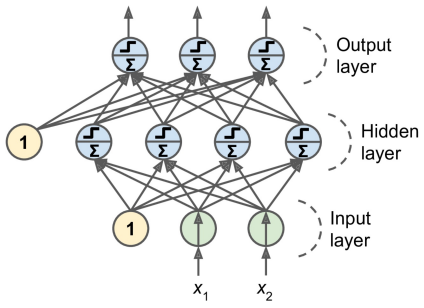


Figure 2.2: A *multilayer perceptron*. Source: (Géron 2017)

NN, a concept from the biological neurons is adapted. D. O. Hebb suggests in his book - The Organization of Behavior: A Neuropsychological Theory (1949) - that neurons that communicate more often reinforce the connection between each other. A useful analogy is that we, humans, tend to get better at things that we practice more. Hebb's idea, or *Hebb's rule* is the idea behind the training algorithm used even today in neural networks.

The algorithm that is used to this day is called *backpropagation* (Rumelhart, Hinton, and Williams (1985)). In short, the algorithm used in deep NNs, consists of two main steps: a *forward pass*, and a *backward pass*. During the forward pass, the input is passed through the network and all the sums and activation function (e.g. a step function) outputs are calculated until the network produces an output. An error, or a loss, is calculated as a function of the prediction and the true value. In the backward pass, the weights of the connections that produced an answer closer to the correct one are strengthened based on the loss function.

How each weight is updated after each iteration is presented in the formula below.

$$w_{i,j}^{(next)} = w_{i,j} - \eta \times \frac{\partial E_{total}}{\partial w_{i,j}},$$

where $w_{i,j}$ is the weight of the connection between neuron i and neuron j. η is the training rate. $\frac{\partial E_{total}}{\partial w_{i,j}}$ represents how much $w_{i,j}$ contributes to the total loss E_{total}.

NNs can be utilized to perform many tasks. However, there are some problems that arise when using that technology. The more intricate a task is, the more neurons and layers need to be added. This increases the size of a model and increases its complexity. Performing backpropagation on a growing number of neurons increases the training time, as well as the requirements for memory and computational power. To handle the increasing computational load, we need to scale our systems.

2.2 Distributed training

It is possible to scale systems vertically or horizontally. Vertical scaling or scaling up implies adding more resources to a single machine. For example - adding more RAM or adding more hard drives to a computer to make it more capable of handling bigger loads. Scaling up has the benefit of paving a simpler course of action because usually no additional work needs to be done on existing software solutions. However, that approach has its limitations, as scaling is limited by either hardware or price issues.

An alternative approach to vertical scaling is the horizontal scaling or scaling out. The concept in short is to link "weaker" machines into a collectively working cluster to perform a task together in a parallel manner. It allows for the addition of new resources in the form of other devices. On the one hand, it is possible to achieve more computational power. On the other, the distributed nature of the the system brings additional issues such as synchronization, fault tolerance, communication, and coordination among others. This adds complexity as a trade-off to the additional power available. In the domain of NNs scaling horizontally is essential as it enables us to use large models that are able to perform complex tasks while operating on big datasets.

Provided we are working in a distributed environment, according to Mayer and Jacobsen (2019), there are several possible approaches to parallelising and distributing an NN model.

Data parallelism

In data parallelism the model is replicated on each machine of a cluster. The dataset is then divided into disjoint subsets, and each is passed to a worker (see Figure 2.3).

The main advantage of data parallelism is that it is model agnostic. It can easily be applied without knowledge or understanding of the domain and the specifics of a model. This approach scales well if the number of parameters in a model are low. This leads to the main disadvantage of data parallelism. Synchronization of parameters between workers can become a huge bottleneck (Krizhevsky (2014), Jia, Zaharia, and Aiken (2018)). Every machine calculates the gradients of its part of the data. After each pass, the machines need to communicate their calculations.

The main approaches are either a *centralized parameter server* (H. Cui et al. (2016), Abadi et al. (2016)), or a *decentralized all-reduce* communication (Zhao and Canny (2013), Sergeev and Del Balso (2018)). There is still ongoing

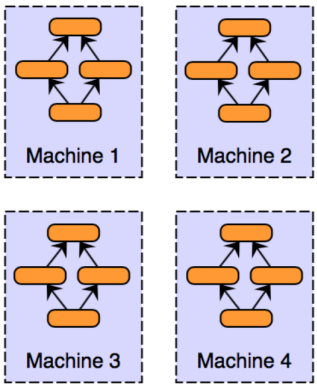


Figure 2.3: Data parallelism. Source: (Xiandong 2017)

research in the area and a lot of effort has been put to tackle this issue. Mayer and Jacobsen (2019) has compiled the most recent advancements. Since this part of the field is so dense, this thesis is not going to focus on data parallel approaches.

Model parallelism

Model parallelism is characterised by the division of the model itself, rather than the dataset (see Figure 2.4). Each machine is responsible for a different part of the model's structure. The machines that hold the input layer are then fed with the data. The output signals are propagated to the machines that hold the parts of the next layer and so on. During the backward pass, workers communicate in reverse order.

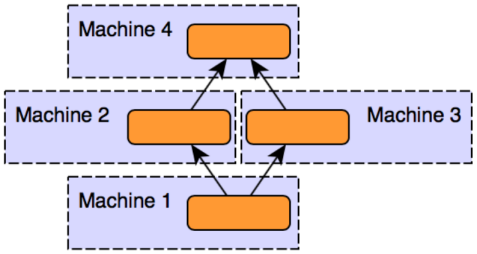


Figure 2.4: Model parallelism. Source: (Xiandong 2017)

The main advantage of this approach is the reduced need for resources. When the model is too big to fit on a single machine, this is the preferred approach. It is possible to take advantage of different devices for performing different tasks in the most optimal way. The reduced requirements for resources,

however, are traded-off for larger amounts of communication. In model parallelism the communication overhead is bigger than the overhead in data parallel methods and is the main disadvantage of this approach.

A major challenge for parallelising an NN model is how to divide and distribute the model throughout a heterogeneous cluster of machines. In other words, which parts of an NN model are placed on what device of the cluster in order to achieve optimal training and inference times. The current approaches focus on the use of NNs, graph embeddings and RL to decide how the model should be split. Efforts from Mirhoseini, Pham, et al. (2017), Mirhoseini, Goldie, et al. (2018), Addanki et al. (2019), and others have been made in the direction of automating the model placement on different devices. This is the **device placement problem**. The device placement problem is the main focus of this work.

2.3 Reinforcement Learning

Machine Learning (ML) is a field of Computer Science that aims to create systems to learn automatically from data. ML algorithms are trained in order to improve their performance in different tasks. The range of applications for ML algorithms is huge and extremely varied and spans across various fields. Some examples include spam filters for emails (Guzella and Caminhas (2009)), natural language processing tasks (Young et al. (2018)), credit card fraud detection (Awoyemi, Adetunmbi, and Oluwadare (2017)), handwriting recognition (Abu Ghosh and Maghari (2017)), etc. ML algorithms are separated into three main categories: supervised learning, unsupervised learning, and RL.

In *supervised learning*, the system tries to learn a predictive model by using data with known outcomes. In other words, the training data is labeled and used as ground truth, and as a result the model makes predictions about unseen data points. An example is a spam filter. A spam filter algorithm is trained using emails labeled as either spam or not spam. Later that system can flag new emails accordingly.

In contrast to supervised learning, *unsupervised learning* algorithms are used when the information is not labeled, and thus no ground truth is provided. The advantage is that the unsupervised models can find complex or hidden connections and structures between the data points. Such algorithms are used to group data into different clusters or to find outliers and anomalies.

Sutton and Barto (2018, p. 1) introduce the concept of *reinforcement learning* (RL) as "learning what to do - how to map situations to actions - so as to maximize a numerical *reward signal*." In RL, an agent learning how to per-

form a task is not given a specific way to achieve the results. Rather, it must discover on its own which actions bring the most reward. Even though there is an obvious similarity between RL and unsupervised learning, in the fact they both do not need a model to operate, there exists a fundamental difference. The goal of RL algorithms is to maximize a reward signal, and not to find a hidden structure in the data. A good application of RL would be a system that needs to respond to an actively changing environment.

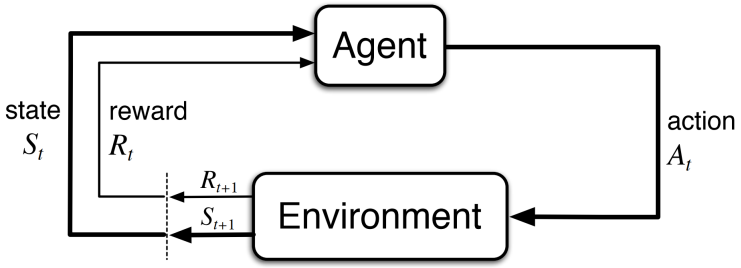


Figure 2.5: Agent-environment interaction. Source: (Sutton and Barto 2018)

There are several important concepts in RL. The *agent* is the part that "thinks", i.e., learns and makes decisions. It interacts with the *environment* that is everything outside its scope. At each time step t the agent receives some representation of the current state of the environment S_t, called *state*. Based on that state, the agent performs some *action* A_t. At the next time step $t+1$, the agent obtains a reward R_{t+1} from the environment, as well as a new state S_{t+1}. The actions of the agent are governed by a policy π, which is a mapping from a given state to an action. The interaction between the environment and an agent is comprised of a sequence of States, Actions, and Rewards $(S_1, A_1, R_1, S_2, A_2, R_2, ..., S_T, A_T, R_T)$, where the sequence ends at a terminal state S_T. Such a sequence from time period 1 to the terminal state at time T is called an *episode*.

On top of that, an agent might have a *model* of the environment. The model is an optional element of an RL system that helps the agent with planning further actions, as it can simulate how the environment would respond in some cases. The goal of the agent is to maximize the reward signal provided by the environment based on its actions. Formally, this process is known as *Markov Decision Process* (MDP) (see Figure 2.5). MDP is an idealized form of the problem of learning from an interactive environment.

There are many RL algorithms as can be seen in the taxonomy presented in Figure 2.6. However, we will not go into detail into most of them. We recommend "Reinforcement learning: an introduction" which is a great book

to dive deeper in RL.

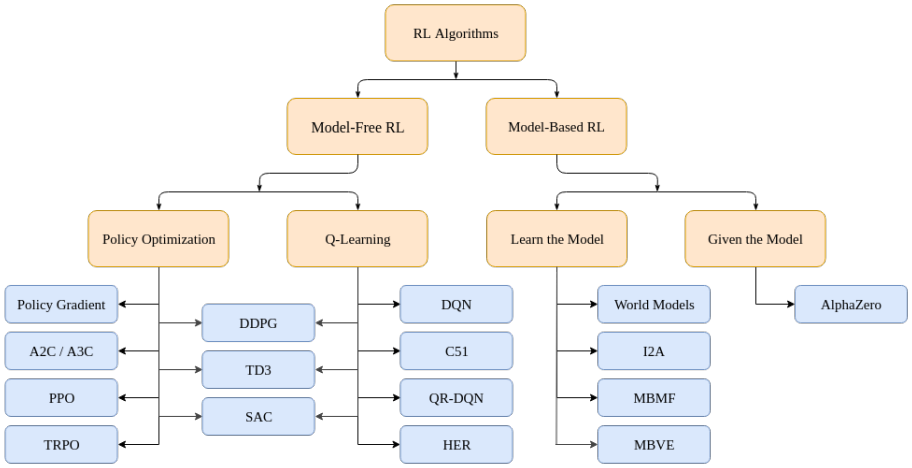


Figure 2.6: RL Algorithms taxonomy. Source: *Part 2: Kinds of RL Algorithms — Spinning Up documentation* (2020)

Almost all RL algorithms involve estimating *value functions* - functions of state-action pairs that estimate how good it is to perform a given action in a given state. *Policy Optimization* methods, by contrast, aim to learn a *parametrized policy* and thus not needing to learn a value function. We use θ for the parameter vector of the policy, and denote the parametrized policy as:

$$\pi(a|s, \theta),$$

where a is an action and s is a state. Policy Optimization methods try to learn θ, based on some performance measure $J(\theta)$, dependent on θ. Maximizing performance is the goal, so these methods approximate *gradient ascent* in J:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)},$$

where $\widehat{\nabla J(\theta_t)}$ is "a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument θ" (Sutton and Barto (2018)).

Current state of the art of device placement solutions (described later in Section 2.5) mainly focus on *REINFORCE* as a Policy Optimization method (Williams (1992)).

REINFORCE

REINFORCE estimates a reward using sampled completed episodes to update the policy θ. The flow of the algorithm is the following (Weng (2018)):

1. θ is initialized randomly
2. Generate a complete episode using policy

$$\pi_\theta : S_1, A_1, R_1, S_2, A_2, R_2, ..., S_T, A_T, R_T$$

3. For $t = 1, 2, ..., T$:
   (a) Estimate the reward G_t
   (b) Update the policy:

$$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta ln\pi_\theta(A_t|S_t),$$

where α is a hyperparameter, and γ is a discount factor.

REINFORCE is explained in detail in Sutton and Barto (2018).

2.4 Graph Embeddings

According to the Mariam-Webster dictionary[1], a graph is a a collection of vertices and edges that join pairs of vertices. Indeed, in mathematics, a graph is a structure that maps pairs of points (vertices) with connections (edges).

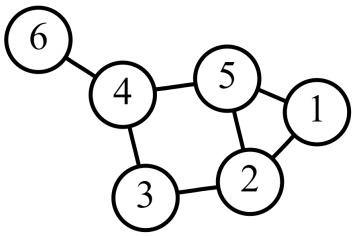


Figure 2.7: A graph. Source: Wikipedia, *Graph theory* (2020)

Now that the basic fundamentals of NNs are explained in Section 2.1, it is easy to see how NN (see Figure 2.2) can be considered as a graph (see Figure 2.7). This fact is one of the pivotal points for the approaches to tackling the device placement problem in general.

Graphs are used to represent data and relationships in many areas such as social graphs of people, linguistic relationships, biological networks, etc. It is possible to do analysis or run various algorithms (for example ML ones) on

[1]Mariam-Webster *Definition of graph* (2020)

the whole graph. A way to represent a graph and its relationships is to use a so-called *adjacency matrix*. The adjacency matrix is a matrix of size $|V| \times |V|$, where V is the number of nodes in the graph. If there is a connection between two nodes, the corresponding element of the matrix will have a non-zero value. Using such a matrix for large graphs quickly becomes infeasible. For example, a graph with only one million nodes would have a matrix with a size of 10^{12}.

This is where graph embeddings come into play. Graph embeddings are compressed vector representations of graphs with lower dimensions. They try to preserve the topology of the graph, as well as the relationships between nodes. Nodes that are similar, tend to be closer together in the target vector space. Some applications of graph embeddings are node classification (surveyed in Bhagat, Cormode, and Muthukrishnan (2011)), link prediction (surveyed in Liben-Nowell and Kleinberg (2007)), clustering (White and Smyth (2005)), and compression (Feder and Motwani (1991), Navlakha, Rastogi, and Shrivastava (2008)). To formalize what is a graph embedding, the definition from Goyal and Ferrara (2018) is used:

Graph embedding: Given a graph $G = (V, E)$, a graph embedding is a mapping $f : v_i \rightarrow y_i \in \mathbb{R}^d, \forall i \in [1, |V|]$ such that $d \ll |V|$ and the function f preserves some proximity measure defined on graph G.

Preserving proximity means that if nodes v_i and v_k are close or similar to each other in G, then they will be close or similar in the embedding space $\mathbb{R}^d$.

Graph embedding is a relatively new field of research, especially when compared to the NNs. There are several notable directions of how researchers tackle the graph embeddings problem as described by Goyal and Ferrara (2018). The first direction is *factorization based* methods. This approach represents a graph in the form of a matrix, and then factorizes it into a vector. This is the first area of research dating to the early 2000s (Roweis and Saul (2000), Belkin and Niyogi (2001), Ou et al. (2016)).

The second direction is based on the *random walk* methods. As it is stated in the name, the method relies on random walks over the graph, thus, allowing for acquiring node similarity and node centrality (Perozzi, Al-Rfou, and Skiena (2014), Grover and Leskovec (2016)). The third direction, and the most relevant one to this thesis, is to use *deep learning* and deep NNs to learn non-linear dependencies between nodes to generate graph representations. These methods use NN architectures, such as Variational Autoencoders (Wang, P. Cui, and Zhu (2016), Cao, Lu, and Xu (2016)), and Graph Convolutional Networks (GCN) (Kipf and Welling (2017)).

The papers, highlighted below, are the most recent state-of-the-art solutions. One of their traits is that they can *inductively* represent graphs, meaning

that they can generate an embedding for unseen nodes and graphs. Previous works were mainly *transductive* - only focus on working with static and unchanging graphs. This is important for this thesis, as the method could speed the device placement and work on different models without the need to be trained over and over.

GraphSAGE

As mentioned above, *GraphSAGE* (Hamilton, Ying, and Leskovec (2018)) is an inductive method to create graph embeddings, whereas previous methods are mostly transductive. Transductive in this context means that they operate optimally only on static graphs. In other words, those methods were not able generalize and predict labels for graphs that they had not been trained on.

However, in many modern scenarios, graphs are not static and are constantly changing. That is why an inductive method is required. Such an approach allows for training the algorithm on a subset of a graph to create an embedding, which is broad enough to represent the whole graph. Given that general embedding, it is possible to create embeddings for unseen nodes without additional training. Moreover, labeling unseen nodes of that graph is also possible.

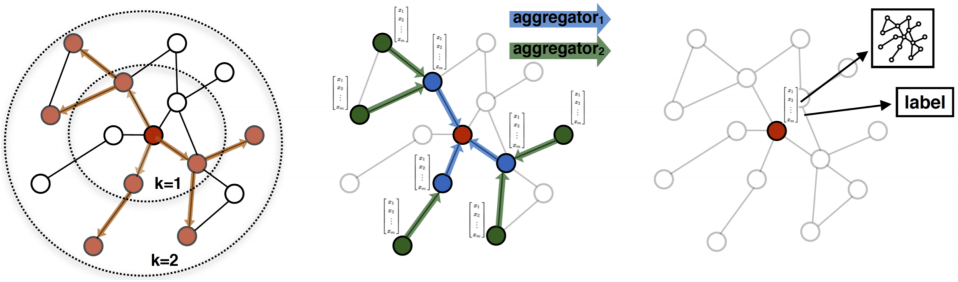


Figure 2.8: *Left:* neighbourhood sampling, *Middle:* Neighbourhood feature aggregation, *Right:* GraphSAGE embedding generation process. Source: Hamilton, Ying, and Leskovec (2018)

Algorithm 1 presents the method used to achieve these capabilities. In essence, the algorithm consists of creating an embedding for each node based on its own features (such as text attributes, node profile information, node degrees) in combination with the aggregated features of the nodes in its neighborhood. The neighborhood of node v consists of layers of the neighbors of the current node up to a distance of K. Each layer consists of a sample of nodes with the same distance to v. The immediate neighbors of v are located

$K = 1$ hops away; their respective descendants are at a distance of $K = 2$ hops; and so on. Note that as aggregations might be computationally expensive, the samples of each node include only a subset of their respective immediate neighbors. Figure 2.8 (*Left*) shows a sample neighborhood of a node with maximum distance of $K = 2$.

The next phase in the algorithm is the formation of the embeddings. The embedding is the concatenation of the features of a node with the aggregation of the embeddings of its own immediate neighborhood. Given a neighborhood of a node v, the feature aggregation starts with the nodes located K hops away from v ($\mathbf{h}_v^0$). Since they do not have further neighbors within the current neighborhood, their embedding is only their own feature vector.

Nodes at the next level (v_{k-1}) form their embeddings by combining the features of their neighbors from the previous level using an aggregation function ($\texttt{AGG}_k(h_u^{k-1}, \forall u \in$ neighborhood of v_{k-1})) and concatenating them to their own features at the current level. This concatenation is multiplied by a weight matrix $\mathbf{W}^k$ and passed through some non-linearity - σ. The process is repeated K times, until node v has all the information needed to form its own embedding. Figure 2.8 (Middle) provides a visual example of the feature aggregation described above.

The weight matrices $\mathbf{W}^k$ are separate for each layer in order to propagate information between different layers. At training time the parameters of the weight matrices are updated. During inference time - node labels and embed-

dings are predicted based on the learned aggregators.

Algorithm 1: GraphSAGE algorithm

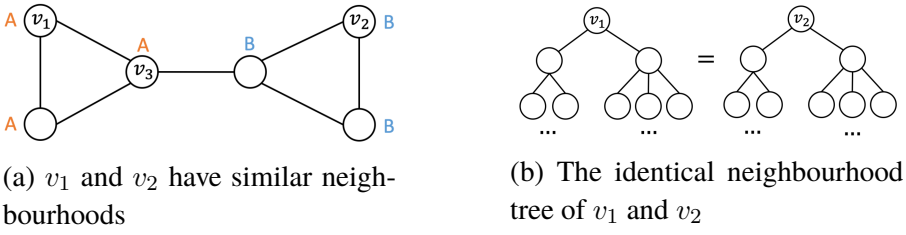
Input: Graph $G = (V, E)$; node input features $\{\mathbf{x}_v, \forall v \in V\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in K$; non-linearity σ; differentiable, order invariant aggregation functions $\texttt{AGG}_k, \forall k \in K$; Neighbourhood function N: $v \rightarrow S$ where S is some subset of neighbors of v.

Output: Graph embeddings $\mathbf{z}_v, \forall v \in V$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in V$;
for $k = 1...K$ **do**
  for $v \in V$ **do**
    $\mathbf{h}_{N(v)}^k \leftarrow \texttt{AGG}_k(\{\mathbf{h}_u^{k-1}, \forall u \in N(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \texttt{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{N(v)}^k))$;
  end
  $\mathbf{h}_v^0 \leftarrow \mathbf{h}_v^0 / \|\mathbf{h}_v^0\|_2, \forall v \in V$;
end
return $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in V$

P-GNN

Positional aware Graph Neural Networks or P-GNN (You, Ying, and Leskovec (2019)) is a part of the current movement in graph embedding research that revolves around Graph NNs. Using graph NNs has the advantage of good performance and inductive capabilities. GraphSAGE (Hamilton, Ying, and Leskovec (2018)) is an avid example of GNNs, however, it has its own limitations. The main limitation is that such methods, and GraphSAGE as an example, fail to catch the positioning and location information of a node, as they only capture its local neighbourhood. This becomes a problem in the case when two nodes with the same neighbourhood structure, located in different parts of a graph, end up in an identical place in the vector space of the graph embedding. For example, in Figure 2.9a, vertices v_2 and v_2 have the same neighbourhood tree. The subtree is shown in Figure 2.9b.



(a) v_1 and v_2 have similar neighbourhoods

(b) The identical neighbourhood tree of v_1 and v_2

Figure 2.9: Source: You, Ying, and Leskovec (2019)

The authors of P-GNN tackle this issue by introducing sets of nodes within the graph, called *anchor sets*. In addition to the neighbourhood aggregation used in GraphSAGE, the nodes additionally aggregate information from the anchor sets. This additional information is weighed based on the distance of the node to the anchor set. This allows for the embedding to also capture the locality of the node in relation to the whole graph. In the case of the example in Figure 2.9, vertex v_3 is introduced as an *anchor set*. The shortest distance between v_1 and v_3 is different than the shortest distance between v_2 and v_3. Thus, in the embedding space, v_1 and v_2 can be distinguished.

Algorithm 2 shows in more detail how the P-GNN approach works. The first step is to calculate the anchor sets S for the current graph G. In the paper, the authors choose $log^2(|V|)$ number of anchor sets. The sets come in different sizes. The reason for that is that only choosing small anchor sets provides very good locality information. However, not all nodes can be covered by those anchor sets. On the other hand, choosing only large anchor sets solves the coverage issue, but does not provide good locality data, as many nodes are covered by it. Choosing anchor sets with varying sizes balances this trade-off and leads to a combination of good coverage and more precise locality information. Note that in the paper the anchor sets are re-chosen at each forward pass of the algorithm.

Algorithm 2: P-GNN algorithm

Input: Graph $G = (V, E)$; node input features $\{\mathbf{x}_v, \forall v \in V\}$; Set S of k different-sized anchor-sets $\{S_i \subset V, i \in 1..k\}$; Message aggregation function $F : x \rightarrow r$-dimensional message; weight matrix $\mathbf{W}$; non-linearity σ; order invariant aggregation function AGG

Output: Graph embeddings $\mathbf{z}_v, \forall v \in V$

$\mathbf{h}_v \leftarrow \mathbf{x}_v, \forall v \in V$;
for $v \in V$ **do**
  $\mathbf{M}_v = \mathbf{0} \in \mathbb{R}^{k \times r}$;
  for $i = 1 \dots K$ **do**
    $M_i \leftarrow \{F(v, u, \mathbf{h}_v, \mathbf{h}_v), \forall u \in S_i\}$;
    $\mathbf{M}_v[i] \leftarrow \texttt{AGG}(M_i)$;
  end
  $\mathbf{z}_v \leftarrow \sigma(\mathbf{M}_v \cdot \mathbf{W})$;
end
return $\mathbf{z}_V \leftarrow \mathbf{z}_v, \forall v \in V$

After the sets have been chosen, an embedding is calculated for each node v and its relation to all nodes $u \in S_i$, where $i \in 1 \dots k$, and k is the number of anchor sets. The relation between v and u is calculated using a function F that combines features of both nodes, as well as locality information, such as distance. The final embedding is formed from a vector $\mathbf{M}v$ that is a vector of size k. Each element M_i corresponds to an aggregation of the relation of v to each node u for each anchor set S_i. $\mathbf{M}v$ is multiplied by a weight matrix $\mathbf{W}$ and is passed through a non-linearity σ to produce the final embedding z_v for node v.

2.5 Device Placement

As previously mentioned, device placement is the study of parallelizing an NN model across a cluster of heterogeneous machines when the model does not fit on a single machine.

We have taken a look at what is the problem and we have described all underlying technologies and methods used in the state-of-the-art solutions. More concretely, we have described what are NNs. Then, we described what are graph embeddings and what is their relation with NNs. Finally, we have briefly described what is RL. In this section we describe how these components are

combined to tackle the device placement problem.

The field is quite new and only recently has become a hot topic for researchers with one of the first works being Mirhoseini, Pham, et al. (2017). There is a clear outline on the basic approach of how scientists tackle the issue. First, an input NN, represented as a graph is separated into groups for optimization purposes. Next, embeddings for the input graph created from those groups are made. The embeddings are then passed to an NN that is used as a *classifier*. The output of the classifier indicates which group should be placed on which device. The resulting placements are used as the state to an RL policy that tries to reduce the runtime of the input NN.

In the rest of this section we introduce some of the state-of-the-art solutions. Namely, we go through one of the first efforts - Mirhoseini, Pham, et al. (2017). We then go through the continuation of that work - Mirhoseini, Goldie, et al. (2018). Next, we describe Gao, Chen, and B. Li (2018), followed by Nazi et al. (2019). Finally, we describe the latest work in the field - Addanki et al. (2019).

Device Placement Optimization with Reinforcement Learning

One of the first major works was done by Mirhoseini, Pham, et al. (2017). They propose a method that learns how to optimize the device placement of TensorFlow graphs using RL. To achieve this, they use a Sequence-to-Sequence *Recurrent Neural Networks* (RNN) (Jain and Medsker (1999)) model that predicts the optimal placements of operations on different devices. An overview of the architecture, as presented in the original paper, is shown in Figure 2.10.

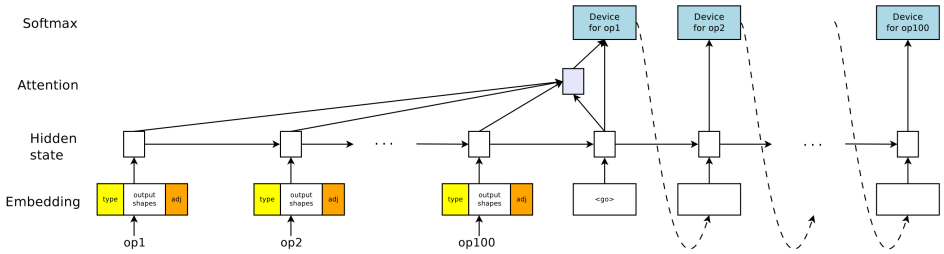


Figure 2.10: Device placement optimization with Reinforcement learning architecture. Source: (Mirhoseini, Pham, et al. 2017)

In essence, they have an *encoder* RNN model and a *decoder* RNN model. The encoder receives information about the operations and dependencies (i.e. nodes and links) between them (as concatenated embeddings for the type of operation, output shape, and one-hot encoded adjacency information), after

which a proposal for placement for each device is predicted by the decoder. Each placement is consequently executed on the actual hardware environment. A reinforcement function based on runtime execution is used to reward good placements (see Section 2.3).

To optimize training and avoid vanishing/exploding gradients in the RNN models, the authors manually co-located certain operations on the same device based on different heuristics. A downside of this method is that it is limited to small graphs (less than around 1000 nodes).

A Hierarchical Model for Device Placement

"A Hierarchical Model for Device Placement" from Mirhoseini, Goldie, et al. (2018) steps on the previous work from Mirhoseini, Pham, et al. (2017). It presents a model for automatic placement of operations on distributed, heterogeneous devices.

To achieve the placements, a Sequence-to-Sequence model called *placer* is introduced. It is similar to the one presented in Mirhoseini, Pham, et al. (2017). Again, a reinforcement function is used to reward the placement prediction model. The reinforcement function calculates the reward based on the execution time of the predicted placement in the actual hardware environment.

The main difference between the two papers is that the current one proposes an end-to-end solution that does not require human intervention. The team added a new submodel called *grouper*. In Mirhoseini, Pham, et al. (2017), the co-location of operations and their grouping is performed manually using various heuristics, which are not scalable. The grouper, on the other hand, replaces the need for manual intervention with a dense network using a softmax layer at the end with an output size equal to the number of groups.

As seen in Figure 2.11, the input to the grouper is the same concatenated vector as used in the paper by Mirhoseini, Pham, et al. (2017) (three concatenated vectors: count of each operation in the group, count of total output shapes in a group, and group one-hot encoded adjacency information). As an output, it uses a softmax layer to predict which group an operation belongs to. The input to the placer's sequence-to-sequence model is an embedding for each group. The overall model output is a device placement for each group (unlike Mirhoseini, Pham, et al. (2017), where the output is a placement for each operation).

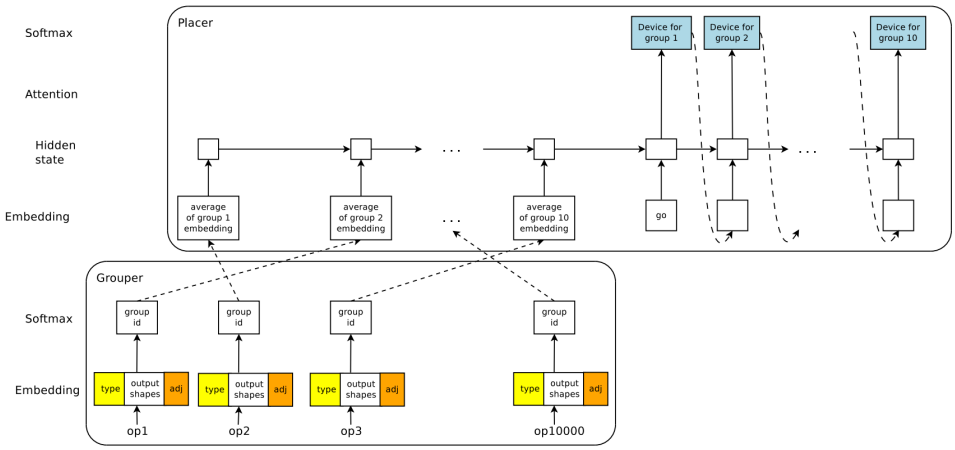


Figure 2.11: A hierarchical model for device placement architecture. Source: (Mirhoseini, Goldie, et al. 2018)

Spotlight

Spotlight (Gao, Chen, and B. Li (2018)) is another RL approach, similar to Mirhoseini, Pham, et al. (2017) and Mirhoseini, Goldie, et al. (2018). The similarity is that they are using an *Long Short-Term Memory* (LSTM) (Hochreiter and Schmidhuber (1997)) layer to produce device placements per group of devices. There are a few differences for this paper in comparison to Mirhoseini, Pham, et al. (2017).

First, in addition to the LSTM architecture, a content attention mechanism is added before the output. Second, the Gradient Policy is calculated using PPO (see Section 2.3), which is a more modern RL approach, compared to the one used previously. In order to apply it, however, the device placement problem needs to be modeled as an MDP. This is an iterative decision process, in which a device is placed at time t, and the next device is placed with the previous steps in mind. An overview of the architecture is presented in Figure 2.12.

Spotlight operates by first mapping NN operations to a dictionary with the name of the existing operators. The set is then fed to the RNN and at each time step an assignment is produced for each set item. Unlike Mirhoseini, Goldie, et al. (2018), the grouping is not done using the default TensorFlow functionality but rather on a newly defined heuristic.

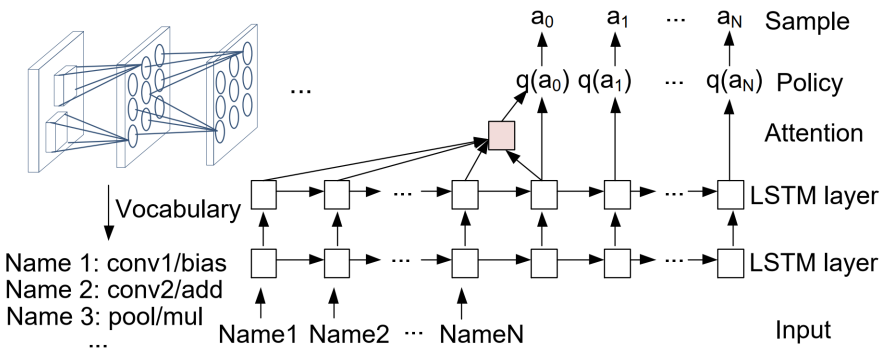


Figure 2.12: Spotlight: Optimizing Device Placement for Training Deep Neural Networks. Source: (Gao, Chen, and B. Li 2018)

GAP: Generalizable Approximate Graph Partitioning Framework

In this paper, the authors propose a generalizable model for graph partitioning. Their approach uses unsupervised learning with an NN to learn optimal partition probabilities. A key achievement is that the algorithm can be taught on small graph sets, but at the same time works well on big (with up to 20000 vertices) unseen graphs with no assumptions about the graph structure.

One of the base elements of the model is the *embedding module*. It is used to produce node embeddings that are learned during training by incorporating node features and graph structure (node degrees, features, adjacency information). The second part is the *graph partitioning module*, which is a dense NN with a softmax layer that yields the probabilities for a node to belong to one of the partitions. The embeddings from the embedding module are used as input to the graph partitioning module. This module is trained using the custom differentiable loss function that takes into account the probabilities, node degrees, and the adjacency matrix among other features. Unlike the previous papers, this work focuses on balanced graph partitioning itself rather than optimizing runtime.

Placeto

"Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning" (Addanki et al. (2019)) is the latest proposition for using an RL method for device placement. The key differentiator is that it is the first paper that proposes placement policies that is generalized to a previously unseen family of graphs.

The algorithm works in the following way. First, it is presented with an

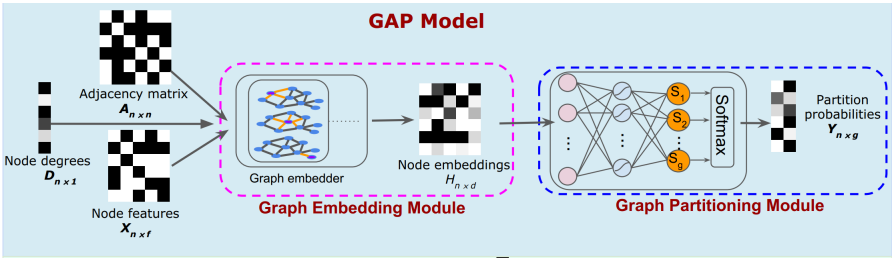


Figure 2.13: GAP: Generalizable Approximate Graph Partitioning Framework. Source: (Nazi et al. 2019)

arbitrary device placement (e.g., all nodes are located on a single device, or are spread out randomly). It traverses all nodes (or node groups, similar to Mirhoseini, Goldie, et al. (2018)). For each node, the algorithm creates embeddings, and based on an RL model, it assigns a placement for that node. After the placement, a reward is calculated using a simulator based on the runtime of the whole graph provided the new placement. That reward is then presented to the next time step, where the next node is considered. Thus, iteratively the algorithm proposes better placements for consecutive nodes using a reward function based on execution runtimes. This is an MDP process, similar to the one used in Spotlight (Gao, Chen, and B. Li (2018)).

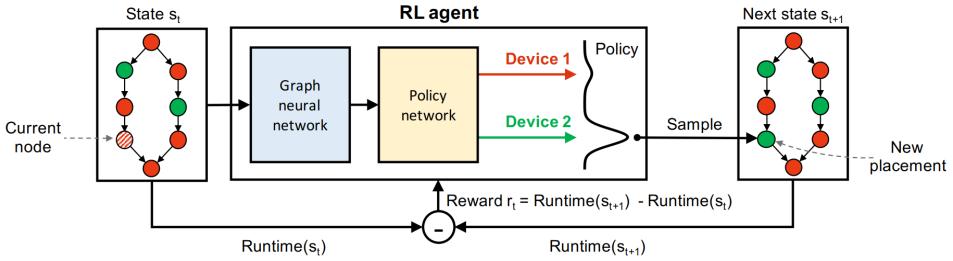


Figure 2.14: Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning Architecture. Source: (Addanki et al. 2019)

The decision model, or RL agent, consists of two main parts: a *graph embedding NN*, and a *policy dense network* (Figure 2.14), such that the latter outputs a device probability for the embedding of the current node, given the current placement. The approach in the model is somewhat similar to the approach in GAP (Nazi et al. (2019)), where the graph embedding is learned. Here the context of a node n is learned by using information about parents (nodes that can reach n via their outgoing connections), children (nodes that can be reached by n via outgoing connections) and parallel nodes (nodes that

cannot be reached by n) using message passing. If the embedding is learned, instead of being a concatenation of graph information (as proposed by Mirhoseini, Goldie, et al. (2018)), it becomes possible to capture general graph structure.

The graph embedding mechanism is presented in more detail in Figure 2.15. First, feature information for a node is collected. The information includes node properties, such as total runtime, output tensor size, and the current placement, among others. Information about the graph from the perspective of the node is also collected by passing messages to all descendants and parent nodes. Moreover, all nodes that are parallel to the current one, i.e. nodes that are not connected, are considered as well. All those components are passed through a dense NN to form the final embedding. That embedding is passed to the next step in the algorithm - namely the policy network.

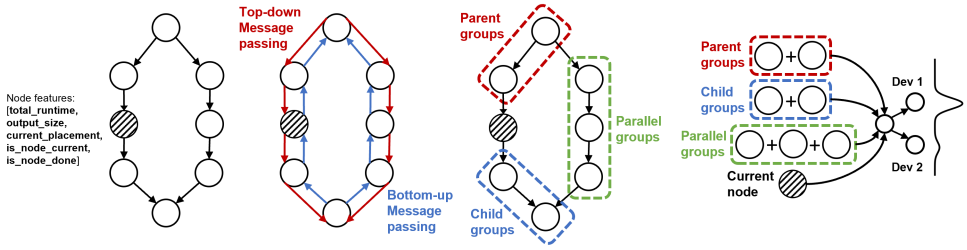


Figure 2.15: Graph embedding information aggregation process of Placeto. Ada[ted from: (Addanki et al. 2019)

At the time of writing, Placeto is the latest published work on the topic. It achieves the best performance and on top of that is able to generalize to previously unseen graphs. One of their contributions is a simulator that allows for measuring performance without the need for actual hardware. On top of that, the authors have provided reference source code that is used as a foundation for this thesis. The graph embedding mechanism is used as a baseline for our contributions, as our work consists of replacing that mechanism with different alternatives.

Chapter 3

Method and Implementation

In this chapter, we describe how we implemented the contributions of this thesis. We go into detail regarding the foundation of this work, as well as how we extended it to reach our goals. First, we describe the Placeto framework, provided as reference code for (Addanki et al. (2019)) and its purpose related to the thesis. Second, we describe what parts of that framework were changed or extended. We then go over some limitations of our work.

3.1 Method overview

In essence, our work aims to explore what is the impact of different graph embedding architectures on current device placement methods.

The lastest device placement methods for NNs use RL to learn placement policies, that require less time than policies created by human experts. Another goal is to reduce the computation time the RL agent needs to produce such policies.

In order for the RL agent to work with an input NN, that NN is usually represented as a graph. That is why different graph embedding methods are explored in various works within the field (as described in Section 2).

Placeto (Addanki et al. (2019)) is the latest work in device placement. We extend that work by replacing its graph embedding module with different state-of-the-art graph embedding methods - GraphSAGE (Hamilton, Ying, and Leskovec (2018)), and P-GNN (You, Ying, and Leskovec (2019)). Figure 3.1 shows more concretely what are our additions to Placeto.

We explore what is the impact of the different graph embedding architectures on the quality of the placement policy and how does it compare to the default baseline implementation of Placeto. We measure the quality by simu-

lating the runtime of the input NN graph representation before and after devising a new placement policy. We also observe how the different architectures affect the computation time required by the RL agent.

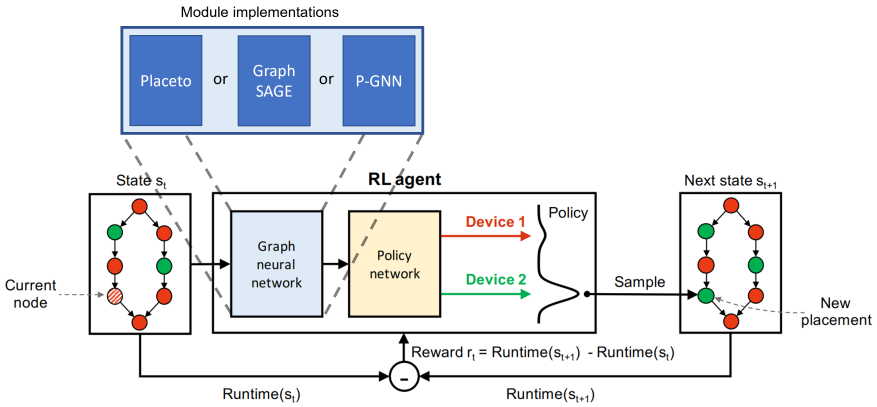


Figure 3.1: This thesis' additions to Placeto. We replace the Graph neural network module with different graph embedding architectures (GraphSAGE and P-GNN), and explore the effects. Adapted from: (Addanki et al. 2019)

3.2 Foundation

As part of the contributions of Addanki et al. (2019), partial source code for "Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning" is open sourced (Addanki (2020)). We are using it as a foundation for our work. Placeto uses an NN based RL agent to parametrize an MDP that optimizes a placement policy. The agent is trained using a Policy Gradient algorithm (see Section 2.3).

The framework works with graph representations. The input graph is preprocessed, such that each node has several features, that are used in the placement process:

- `total_runtime` - total runtime measured in. The information about runtimes is provided as part of the datasets we are using. They are described in more detail in Section 4.1.1.
- `output_size` - the number of edges going out of the current node
- `current_placement` - carries information about the device on which the node is placed currently. This property may change at every iteration, depending on the placement policy

- `is_node_current` - this property bears meta information used to facilitate the MDP
- `is_node_done` - another property with meta information to help run the MDP. When all the nodes are marked as done, the episode finishes.

The graph is passed to an RL agent, whose internal structure is an NN. The architecture consists of two modules, that are logically separated into a graph neural network and a policy neural network. The NN's architecture is described in more details in Section 2.5. For clarity, in Figure 3.2, we show again how Placeto uses the features of the input graph to produce a placement suggestion.

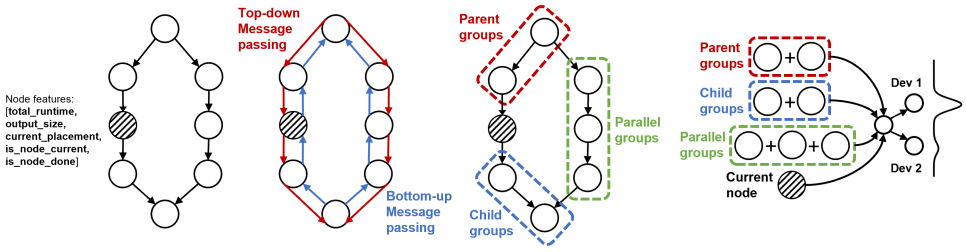


Figure 3.2: Graph embedding information aggregation process of Placeto. Adapted from: (Addanki et al. 2019)

The source code allows for changes to the internal NN architecture of the RL agent. It is possible to build a custom NN for both the graph embedding network and the policy network. Since Placeto uses *Tensorflow* (TF) (Abadi et al. (2016)), all operations in TF are allowed, as long as they abide the rules for input and output tensor sizes. Furthermore, Placeto provides a simple implementation of an MLP called *Feedforward NN (FNN)*, which is used as the building block of the whole NN. Provided input size, output size and hidden layer size, it creates a three-layer TF network with the given dimensions. The biases of each layer are initialized with zeroes, and the initial weights are initialized using the method presented in Glorot and Bengio (2010). The chosen non-linearity function between each layer is *Rectified Linear Unit* (ReLU).

The framework provides an engine for running MDP episodes. It allows for choosing initial placements for the nodes in the graph. It can run episodes of the MDP and update the weights in the NN based on rewards. Placeto can automatically calculate the rewards based on the Policy Gradient algorithm. The parameters used in the algorithm are updated based on the output from the built-in simulator. The simulator can compute with high accuracy the runtime

of an NN. The simulation does not require real hardware, which makes the usage of the whole framework resource effective.

There are mechanisms in place for controlling hyperparameters and monitoring the results.

3.3 Implementation details

To achieve our goal of figuring out how to tackle the device placement problem, we have implemented two different graph embedding network architectures - GraphSAGE (Hamilton, Ying, and Leskovec (2018)) and P-GNN (You, Ying, and Leskovec (2019)). The two architectures are state-of-the-art techniques in graph embedding. For an in-depth description of the algorithms and how they work - see Section 2.4.

Both the GraphSAGE and the P-GNN implementation we have can be used instead of the one already presented in the Placeto framework. They are implemented as modules, and thus they can be easily swapped with the default implementation.

3.3.1 GraphSAGE implementation details

Our implementation of GraphSAGE follows the algorithm presented in Hamilton, Ying, and Leskovec (2018) faithfully. In this section, we will go into more technical details and explain design choices. The logic behind GraphSAGE is that an embedding of a node is constructed using that node's mean aggregated neighbourhood information, concatenated with its own features. The aggregation of the neighbourhood of a node encompasses nodes located at a distance of up to two levels (the number is configurable).

Node embedding

Initially, we pass each node's features through an FNN (see Section 3.2) to create the initial node embeddings. The node features that we use are the same as the ones used by Placeto and are described in Section 3.2. The features are:

- `total_runtime`
- `output_size`
- `current_placement`
- `is_node_current`

- `is_node_done`

The random sample of neighbours for each node is chosen as well.

We then proceed to build the subsequent embedding levels using the embeddings we have already created. A node's embedding is calulated using the formula:

$$\texttt{node}_{\texttt{lvl=i}} = \texttt{FNN}(\texttt{CONCAT}(\texttt{node}_{\texttt{lvl=i-1}}, \texttt{AGG}(\texttt{neighbour}_{\texttt{lvl=i-1}}))),$$

where $\texttt{node}_{\texttt{level=i}}$ is the node embedding of node u at level $i \in [1, n]$. $\texttt{neighbour}_{\texttt{lvl=i-1}}$ is the embedding of the immediate neighbourhood of node u at level $i - 1$. `AGG` is an order invariant aggregation function, such as `mean` or `sum`. `FNN` is a 3 layer MLP, which serves as a non-linearity.

Figure 3.3 shows an example of how the graph embedding module is used to help produce a device placement suggestion. On the left side, we see a sample graph, where the current node is marked. Its features are described as well. In the middle section, we see how the neighbourhood embedding of the current node is aggregated from neighbours 2 hops away. Lastly, on the right side of the figure, we see that the embedding is the combination of the current node and the aggregated information from the neighbourhood. That information is passed to the Policy Network, which outputs a device suggestion for the current node.

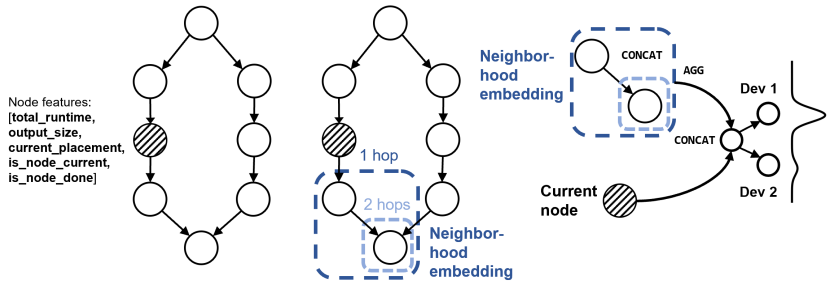


Figure 3.3: Example of how GraphSAGE architecture is used to create node embeddings and produce device placement suggestions. *Left:* Sample graph and node features. *Middle:* Neighbourhood embedding. *Right:* Concatenation of the aggregated neighbourhood embedding.

We initialize different FNNs for each level in the algorithm. For example, if the algorithm is configured to have two hops (as presented in the original paper), then we initialize three FNNs - one for final embeddings of the nodes, and one for each hop away. Using a single instance of an FNN by multiple

nodes means that the weights are shared between the nodes. One of the principles of the GraphSAGE algorithm is that nodes that are *n* hops away share the same weights.

Based on the temporary embeddings of each node's own features, we create the embedding of each node by concatenating those features with the aggregation of the features of its neighbours. The aggregation function we use is `mean`, which is the same as the one used in Hamilton, Ying, and Leskovec (2018). However, we provide the option to use other aggregation functions, such as `max`, `min`, or `sum`.

The concatenation is then stored as the embedding of this node for the next level. We continue this process until we have stored the embeddings for the needed level. We then return a vector containing the embeddings for each node of the graph.

Before adding an embedding for the next level to the list of embeddings, we add a *dropout layer* to the FNN. A dropout layer forces some neurons of the network to be excluded from the calculation of weights. The chance of a neuron being turned off is controlled by a hyperparameter called `droupout rate`. We use dropout layers in order to prevent *overfitting* of the model to the current graph. Overfitting would prevent generalization of the model, and that would, in turn, hinder the performance of the final placement policy when used with unseen graphs. The dropout layer is added for every node embedding at every level of the algorithm, except the last one. It has slight performance impact but improves the produced placement policies.

Partial code of our implementation is shown in Code Block 1.

```
# Generate embeddings for nodes located n hops away.
for i in range(self.hops + 1):
  for n in G.nodes():
    # Get the embeddings of the current node and its
    # neighbours at the current level
    embedding, neighbor_embedding =
      self._get_embeddings(G, n, i)

    # Aggregate the neighbours of the current node.
    neighbor_aggregation =
      self._aggregate_for_node(neighbor_embedding)

    # Concatenate the neighbourhood aggregation
    # with the embedding of the current node
```

```
15      concatenated_with_current =
16        tf.concat((neighbor_aggregation, embedding),axis=1)
17
18      # Generate embedding for the concatenation
19      embedding = self.fnns[i]
20        .build(concatenated_with_current)
21
22      # Add the generated embedding to the dictionary
23      if i != self.hops:
24          self.samples[n][str(i + 1)] = embedding
25      else:
26        embedding = tf.nn.l2_normalize(embedding)
27        self.samples[n][str(i)] = embedding
```

Code Block 1: Function for building GraphSAGE embeddings

We have exposed some controls to manage the behaviour of the algorithm.

- `sample_ratio` - what percentage of a node's neighbours will be taken into account when generating the embeddings of that node. The ratio does not change, no matter which level of embedding we are creating. The default value is 50%, meaning that half of the neighbours will be considered in the calculations.
- `hops` - the number of levels of neighbours from which we are aggregating information. The original paper works with two levels which is also our default.
- `aggregation` - we allow for choosing which order invariant aggregation function can be used to calculate the embedding of a node. The options are: *mean*, *max*, *min*, or *sum*. Our default aggregation function is *mean*, which is also the one used by the authors of the original paper.
- `dropout_rate` - is the hyperparameter used to control how many nodes will be switched off during each iteration of training. Our default is 0.5.

3.3.2 P-GNN implementation details

The second graph embedding architecture we have implemented is based on the work presented in You, Ying, and Leskovec (2019). The main idea behind that paper is that even if a node has an embedding that encompasses the information about a node itself and its neighbourhood, it can still lack locality information. That is why anchor sets are introduced. Anchor sets are sets of nodes integrated into a node's embedding. They allow for differentiation between similar nodes, located at different places in the input graph.

Node embedding

Our implementation of P-GNN uses our GraphSAGE implementation as a base. That means that a node's self-embedding is the same embedding produced by our GraphSAGE function. The P-GNN addition of anchor sets can be seen as an extension to the former. That embedding variant differs from the original implementation of P-GNN; however, it is mentioned in You, Ying, and Leskovec (2019) as a viable option.

We have chosen to step on the GraphSAGE implementation because of two reasons. First, because of the way we have implemented GraphSAGE, it is trivial to use the generated embeddings further down the line. Second, we wanted to investigate what would be the impact of combining those two state-of-the-art works on the final placement policies.

Figure 3.4 gives a visual example of how we use GraphSAGE as a base embedding and then how we add anchor sets to the overall node embeddings. On the left side of the figure, we see how a GraphSAGE type of embedding is created for the current node. Next, we see how two anchor sets of different sizes are randomly chosen - Anchor Set 1 and Anchor Set 2. Lastly, we see how the information of the current node is combined with the information of all anchor sets. The information from the anchor sets is aggregated (described later in this section). After that, it is multiplied with the distance between the current node and the anchor set.

When using the P-GNN architecture, the node embeddings are calculated as previously. However, due to performance issues, the final GraphSAGE embedding's dimension is reduced using a *pooling* layer.

Building anchor sets

After the embeddings for each node are completed, we choose the anchor sets based on the method used in the original paper. An anchor set is a set of

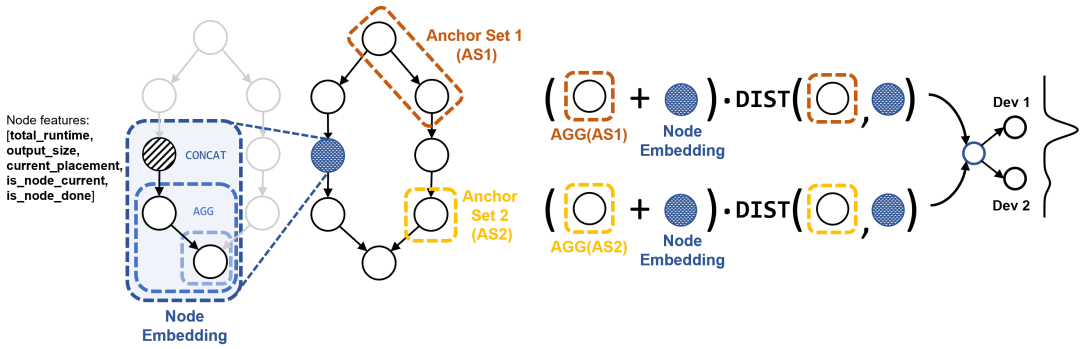


Figure 3.4: Example of how P-GNN architecture is used to create node embeddings and produce device placement suggestions. *Left:* GraphSAGE embedding is created for the current node. Different sized random sets of nodes (anchor sets) - AS1 and AS2 are picked. *Right:* The information between a node and all anchor sets is combined.

randomly chosen nodes from the input graph. The number of sets is $c.log^2(n)$, where c is a hyperparameter. The sets are of varying size. The size is equal to

$$\frac{n}{2^{i+\texttt{anchor\_exponent}}}$$

where n is the number of nodes, $i \in [1, ,log(n))$. The `anchor_exponent` is our addition to provide better control over the size of the anchor sets. It also serves as an optimization function. See Code Block 2 for the actual anchor sets building implementation.

```
def _build_anchor_sets(self, G):
  # G is the input graph
  n = len(G.nodes())
  m = int(np.log(n))

  # `self.pgnn_c` defaults to 0.5
  copy = int(self.pgnn_c * m)

  for i in range(m):
    # self.pggn_anchor_exponent is a mechanism
    # to control the size of the sets
    anchor_size = int(n / np.exp2(i +
          self.pgnn_anchor_exponent))
```

```
14
15      for j in range(np.maximum(copy, 1)):
16        self.anchor_sets.append(
17          random.sample(G.nodes(), anchor_size))
```

Code Block 2: Function for building anchor sets given an input graph

Adding positional information

The next step is to create an embedding for each node with all anchor sets. We create this embedding using the same method as You, Ying, and Leskovec (2019). The combination of anchor set information and node information is done using an aggregation of the anchor set. The aggregation is then concatenated with the embedding of the node. The concatenation is multiplied by the distance between the node and the anchor set. More formally, a relation between a node and an anchor set is the following:

$$\texttt{node\_anchor\_relation} = \texttt{dist}_{(\texttt{n},\texttt{u})}.\texttt{CONCAT}(\texttt{node\_emb}, \texttt{anchor\_emb}),$$

where `n` is the current node, `u` is a node from an anchor set. $\texttt{dist}_{(\texttt{n},\texttt{u})}$ is the distance between the nodes `n` and `u`. `anchor_emb`, or anchor embedding is an aggregation of an anchor set.

We have implemented two types of anchor set aggregations - a mean aggregation, and a max aggregation, that can be used as anchor embeddings. Figure 3.5 provides an example of how the two types of aggregation work. Max aggregation (Figure 3.5 Right Top) calculates the relation between a node and an anchor set by combining the embedding of the current node with the embedding of the furthest node of the anchor set. In this case - node A, as it is located at a distance of 2. Node B is located at a distance of 1 from the current node. Mean aggregation (Figure 3.5 Right Bottom) implies that a relation is calculated between the current node and every single node from the anchor set. The final embedding is the mean of those relations.

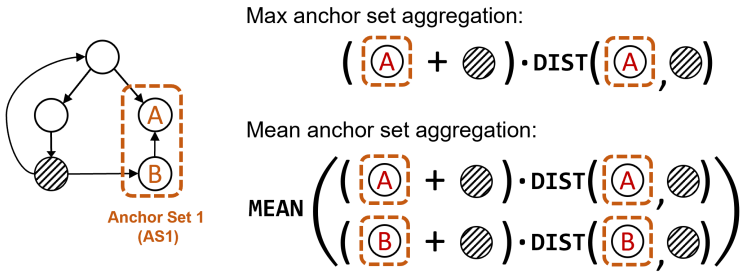


Figure 3.5: Mean and max anchor set aggregation example. *Left:* Sample graph, where the anchor set consists of nodes A and B. *Right (Top):* Max anchor set aggregation. *Right (Bottom):* Mean anchor set aggregation.

The mean aggregation calculates the `node_anchor_relation` for a node n to each node in the anchor set. The final embedding for this anchor set is calculated using a mean aggregation between the resulting `node_anchor_relations`. This approach has a considerable performance impact and is not our default solution.

On the other hand, we have max aggregation. Here we calculate a `node_anchor_relation` between a node n and only a single node u from the anchor set. That node is the one with the maximum distance between n and u. Partial code is shown in Code Block 3. It is equal to the number of edges of the graph in the shortest path between n and u.

When a node has calculated its relation to each anchor set, those relations are aggregated into the ultimate node embedding. Finally, that embedding is passed through an FNN.

```
def _max_aggregate_anchor(self, anchor_set, node):
  # get the node with the maximum distance
  max_agg_anchor = max(anchor_node_intersections)
  node_embedding = self.samples[node][str(self.hops)]
  # get the precalculated embedding of the max node
  anchor_embedding =
    self.samples[max_agg_anchor[0]][str(self.hops)]

  # calculate the distance
  positional_info = 1 /
    (self.distances[node][max_agg_anchor[0]] + 1)

  # concatenate the embeddings of the node
```

```
14  # and the max anchor node
15  feature_info =
16    tf.concat((node_embedding, anchor_embedding),
17      axis=0)
18
19  node_anchor_relation =
20    positional_info * feature_info
21
22  return node_anchor_relation
```

Code Block 3: Function for max aggregation of anchor sets

We have exposed several control levers to manipulate the behaviour of the code.

- `c` hyperparameter - used to control the number of anchor sets
- `anchor_exponent` - used to control the size of the anchor sets.
- `aggregation` - controls how to combine a node and an anchor sets. Possible options are mean and max, with max being the default
- `neighbour_cutoff` - controls how far apart two nodes can be to be considered connected. The default value is 6

All GraphSAGE controls, described in the previous section, apply to the embeddings here.

3.4 Design choices and Limitations

In this section, we make an overview of some of the technical limitations we faced and what decisions were made based on them. Some of them stem from the implementation of the Placeto framework, whereas others are deliberate choices.

First, from a higher perspective, we use the RL and Simulator capabilities of the framework without changing them, accepting their limitations. Our focus is not to prove or disprove the work done in Addanki et al. (2019) (except the graph embedding part), but instead, we have used the provided work as

a stepping stone, based on which we can compare and contrast the different architectures we have implemented.

Second, for the feature representation of graph nodes, we use the same feature set as the one used in Placeto. It contains several different properties, some of which describe the node itself, while others contain metadata. In order to integrate our code with the framework, we comply with its structure and are using it as-is. This has an impact on the input and output dimensions of the sub-networks we have developed.

Third, our implementation of the position-aware architecture is mostly faithful to the source You, Ying, and Leskovec (2019). However, there are several differences. In the paper, the anchor sets are picked at every forward pass of the network. In contrast, we are choosing the anchor sets only once in the beginning. The reason is that Placeto uses an old version of Tensorflow. In it, the computation graph is built lazily at the beginning. In addition to that, our models become large. That leads to a considerable performance impact when trying to rebuild the computation graph with the new anchor sets in mind. We implemented this approach as well, but it rendered testing impossible due to its performance. Tensorflow 2 allows for the eager building of the computation graph, and we would have been able to perform this. The reference code provided by the authors of the paper is written using the *PyTorch* framework (Paszke et al. (2019)), which is an alternative to Tensorflow. Its architecture allows for easy calculation on per iteration basis. However, rewriting the Placeto framework to use Tensorflow 2 or PyTorch is out of the scope of this thesis.

Chapter 4

Evaluation

In this chapter, we describe the outcome of our work. First, we explain our experimental setup. That includes our datasets, what we are comparing and other details that might be useful for the reproducibility of our experiments. We then go to report the results and discuss them.

4.1 Experimental Setup

4.1.1 Datasets

Our work requires us to evaluate different NN models. Our approach needs the NN model to be represented as a graph. That is why we choose to evaluate our work with computation graphs generated by Tensorflow. Tensorflow can create a computation graph of all the operations in the model before performing any computation. That approach is what we use as a base for our input datasets because it transforms an NN model into our desired input - a graph.

We evaluate our work on three different synthetic datasets, comprised of graphs with different sizes and structures. The datasets are the same used to evaluate the generalizability of Addanki et al. (2019). Our goal is not to evaluate the generalizability of Placeto. However, the same datasets provide varied input for our experiments and allow us to assess our work better. In a similar fashion to the paper, we refer to the datasets as *cifar10*, *ptb*, and *nmt* (lowercase, italic).

Two of our synthetic datasets are generated using the *Efficient Neural Architecture Search* (Pham et al. (2018)) (ENAS) system. ENAS is a system that is able to generate artificial NN models using an RL based approach by searching for an optimal subgraph within a larger graph. It is trained to maximize

some reward based on a given dataset. An example is shown in Figure 4.1. On the left side of the figure, there is a graph containing different operations, i.e. *conv 3x3*, *conv 5x5*, *max 3x3*, *sep 3x3*. The RL agent of ENAS can generate different ordered combinations of the operations in that graph. On the right, we see an example of a constructed convolutional NN.

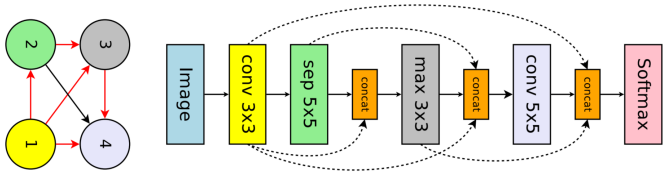


Figure 4.1: ENAS generated convolutional NN architecture. *Left:* graph of operations. *Right:* subgraph of operations created from the graph on the left. Source: (Pham et al. 2018)

The *cifar10* and *ptb* datasets are generated using ENAS system.

Our $cifar10$ dataset is created using ENAS based on the CIFAR-10 (Krizhevsky (2009)) image dataset. The CIFAR-10 image dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. Using CIFAR-10 and classification accuracy as the reward signal to the ENAS controller, multiple convolutional NN models with high accuracy over the dataset are generated. N sized sample of these architectures is randomly chosen to form the $cifar10$ dataset that we are using.

The Penn Treebank language modeling dataset (Marcus et al. (1999)) is the basis for the second graph dataset we have - *ptb*. It contains 2499 stories from a three year Wall Street Journal (WSJ) collection of 98732 stories for syntactic annotation. Using the validation perplexity as a reward to the ENAS controller, recurrent NN models, which are suited for language modelling, are automatically generated by ENAS. N sized random sample of the generated models is chosen to become the provided ptb dataset.

Last, the nmt dataset is based on the NN architecture used in Neural Machine Translation with Attention (Tensorflow (2020)) (*NMT*). It is a sequence to sequence recurrent model using an encoder-decoder architecture (Luong, Pham, and Manning (2015)). N different variations of the NMT model are generated by sampling the number of unrolled steps to produce the Tensorflow computation graphs.

In order to reduce the number of operations (ops) in each computation graph, some operations are grouped into *groups* of similar operations. The approach is similar to the grouping in Mirhoseini, Pham, et al. (2017).

Table 4.1 summarizes our datasets - what is their architecture and what is their average size in terms of node groups. The datasets contain varied amounts of nodes, which provides data on how our work affects different sized graphs.

2*	2***Model architecture**	**# of nodes**	
		Average	σ
cifar10	Convolutional	301	12
ptb	Recurrent	500	47
nmt	Recurrent	188	31

Table 4.1: Dataset summary

4.1.2 Baselines

We compare the two graph embedding architectures we have implemented to the one presented in Addanki et al. (2019). To compare, we use the runtime of the input graph, which represents an NN, calculated by the built-in simulator. All of the Placeto framework settings are set to identical values. We evaluate each of the architectures on the same datasets. For all experiments, the default placement is that all nodes are located on a single device.

To compare, we use the runtime of the input graph, which represents an NN, calculated by the built-in simulator. We evaluate Placeto, GraphSAGE and P-GNN on the same datasets in order to calculate averages in runtime improvement and computation time. We then assess the improvements or decreases in the results of GraphSAGE and P-GNN in relation to Placeto. All of the Placeto framework settings are set to identical values. For all experiments, the default placement is that all nodes are located on a single device.

4.1.3 Training Details

We conduct the experiments on the three different datasets - *cifar10*, *ptb*, and *nmt*. The baseline, i.e. the Placeto default implementation, is used with the framework's default settings, where the graph embedding module uses the aggregation presented in Addanki et al. (2019). Each graph embedding is then aggregated using an NN. The purpose is to funnel the graph embedding into the policy module, that outputs the actual device placement suggestion.

When testing the GraphSAGE graph embedding module, we bypass the Placeto embedding. The embeddings are again aggregated before being fed to the policy module. The P-GNN graph embedding module steps on top of

the GraphSAGE implementation as it uses the GraphSAGE embeddings as its base node embeddings. When testing P-GNN, all GraphSAGE settings are kept the same as when testing GraphSAGE on its own.

The settings for our GraphSAGE module are shown in Table 4.2. These settings are carried over all datasets. Initially, all nodes of the input graph are placed on a single device, and that placement is updated as the episodes progress. The way the experiment progresses is that the best placement from an episode is used as an input to the next one.

	GraphSAGE	**P-GNN**
Initial placement	Single device	
Sample ratio	0.8	
Hops	2	
Aggregation (neighbourhood)	mean	
Droupout rate	0.7	
c hyperparameter	-	0.2
Anchor exponent	-	4
Aggregation (anchor set)	-	max
Neighbour cutoff	-	6

Table 4.2: Training settings for GraphSAGE and P-GNN

We conduct experiments on all three datasets described in Section 4.1.1, namely $cifar10$, nmt, and ptb. For each dataset, the simulator classifies operations across 3, 5 and 8 devices. That adds up to 9 experiments in total. Each experiment runs the simulation over 51 randomly picked input graphs (17 graphs for each dataset). Each input graph passes through 20 episodes. We chose 20 episodes, as the improvements on average reach a plateau after about 10 to 15 episodes.

All of our experiments are conducted on a machine with an AMD Ryzen Threadripper 2920X 12-Core Processor with a max boost clock of up to 4.3GHz. The machine is equipped with a GeForce RTX 2070 SUPER GPU with 8GB of memory.

4.2 Results

In this section, we evaluate how the different graph embedding modules impact the runtime of an NN according to the simulator (which we refer to as *runtime improvement*), as well as the time it takes the RL agent to calculate

the placement suggestions (*computation time*). We describe and compare the following parameters:

- Runtime improvements of a placement using a particular graph embedding architecture compared to the initial state
- Computation time of the RL agent's internal NN in the different experiments
- Relation between a placement runtime improvement and the computation time required to achieve that result.

The section is separated according to the researched metrics. Finally, we summarize the results.

4.2.1 Hypotheses and expectations

Our primary hypothesis is that changing the graph embedding module would bring improvement to the runtime of a placement. The reason is that it should better capture the structure of the input graph. That would lead to a more optimal classification and placement suggestion from the policy module. Since we are comparing three different graph embedding architectures - we are expecting different runtime gains.

First, we hypothesise, that the Placeto graph embedding architecture (Addanki et al. (2019)) would have a similar positive impact on runtime as the GraphSAGE architecture (Hamilton, Ying, and Leskovec (2018)). The reason is that both graph embedding methods are similar in the sense that both approaches collect data from each node's neighbourhood in some way. On the one hand, Placeto collects information for each node's surroundings by combining information about child nodes, parent nodes, and parallel nodes. On the other hand, GraphSAGE collects that information by combining a node's immediate neighbourhood at some distance. Regarding the time it takes the RL agent - we expect to see similar times due to the architectural similarities.

Second, we hypothesise that the position-aware architecture (P-GNN) presented in You, Ying, and Leskovec (2019) would improve on the results achieved by the other two architectures. The reason is that on top of the neighbourhood information contained in the graph embedding, there is information regarding the locality of each node. We expect that to help better represent the graph as an embedding to pass to the policy module. We expect that the runtime improvement would come at the expense of the time it takes the RL agent to

suggest node placements. That is because our P-GNN implementation is an extension to our GraphSAGE one, and includes additional overhead.

Finally, we relate the runtime improvement to the RL agent's NN runtime using the ratio between the average percentage of improvement and the total RL agent runtime: $\alpha.improvement/\beta.nn\_runtime$. The purpose is to show the performance gain and how it might be offset by longer computation time. For brevity we call that ratio the *Performance-Computation Ratio*, or *P-C Ratio*. If there is no improvement, the ratio becomes 0. α and β are hyperparameters, that can be used to give different weights to the components. In our case, the improvement and the NN runtime of the RL agent have equal weights. Thus $\alpha = \beta = 1$.

4.2.2 Runtime improvement

In this section we show the results we achieved regarding the runtime improvements of placement policies using our different graph embedding architectures. We group the experiments into datasets, where each dataset has a dedicated subsection. Each subsection contains the results of all experiments (3, 5, and 8 devices) ran on that dataset. Finally, we present a summary of all runtime improvement results.

cifar10

When running the simulation over the graphs in the $cifar10$ dataset, we see that all graph embedding modules help produce better placement policies. The policies are such that they improve the runtime compared to the initial single device placement. In Figure 4.2, we see an example that runtimes improve after a single episode after passing the input NN through the RL agent. The runtimes are progressively reduced until a plateau is reached around the 10th or 15th episode. Figure 4.2 is representative of the general trend of how runtimes improve over the course of the experiments.

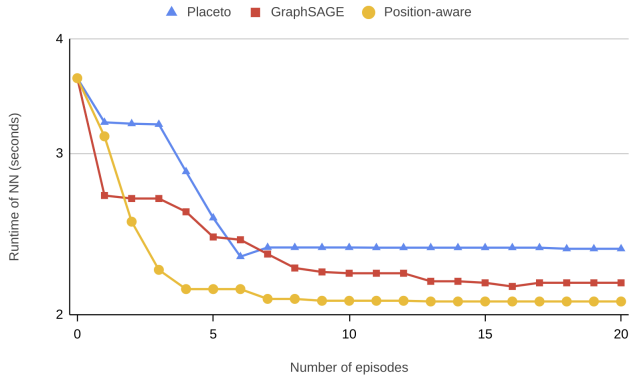


Figure 4.2: Runtime improvement of an input convolutional NN from the $cifar10$ dataset using different graph embedding modules (Placeto, GraphSAGE, and P-GNN) over 20 episodes. Logarithmic scale.

As hypothesised, we see that the position-aware architecture provides the most significant runtime improvement after 20 episodes. This is true for all experiments on the $cifar10$ dataset, namely experiments with 3, 5, and 8 devices. The GraphSAGE architecture performs better than the Placeto one in the experiments with 3 and 5 devices. However, in the experiment with 8 devices, the Placeto architecture brought more substantial runtime improvement.

In the experiment with 3 devices, Placeto provides a significant improvement to the default runtime with 24.585%. The Placeto improvements are less than the improvements provided by the GraphSAGE architecture - 29.566%. P-GNN achieves the most prominent and most consistent improvement of 32.717% (Figure 4.3). It needs to be noted that for some input NNs, the Placeto architecture provides no improvement whatsoever.

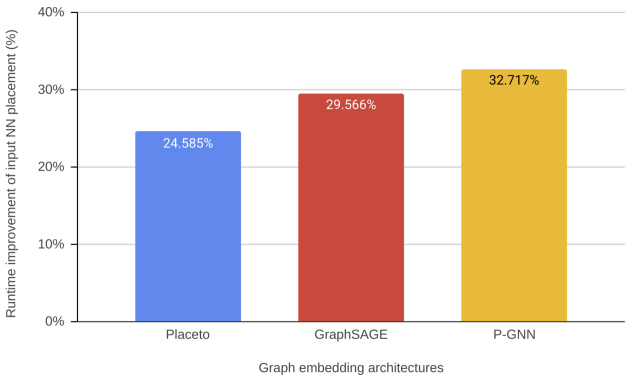


Figure 4.3: Average improvement of the runtime based on suggested placements in %. Experiment is run on the $cifar10$ dataset. Simulator classifies nodes on 3 devices.

Running the experiment on $cifar10$ with 5 devices affirms the trend of P-GNN providing the biggest runtime improvement with an average of 36.084%. The difference between P-GNN and the runner-up in this experiment is bigger than the 3 devices one. GraphSAGE edges over Placeto with 30.567% for the former and 28.250% for the latter (Figure 4.4).

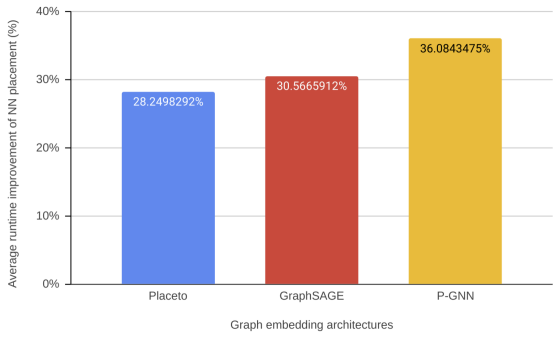


Figure 4.4: Average improvement of the runtime based on suggested placements in %. Experiment is run using the $cifar10$. Simulator classifies nodes on 5 devices.

Lastly, our experiment with 8 devices produces smaller differences between the implementations. P-GNN again shows the best results with 37.114% average runtime improvement. In this experiment, Placeto performs better than GraphSAGE, and its difference with P-GNN is quite small - 2.775%. Placeto averages the runtime improvements to 34.339%, while GraphSAGE scores 31.374% (see Figure 4.5).

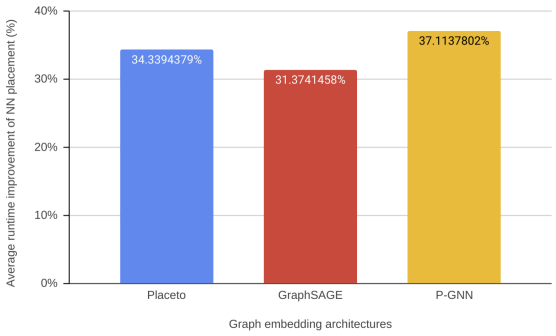


Figure 4.5: Average improvement of the runtime based on suggested placements in %. Experiment is run on the $cifar10$ dataset. Simulator classifies nodes on 8 devices.

In Table 4.3, we summarize the results we achieved when measuring the runtime improvements when using the different graph embedding architectures. The overall average confirms our first hypothesis that P-GNN would achieve the most significant improvements. We also see the expected result that the GraphSAGE and Placeto architectures would bring similar gains. In other words, GraphSAGE does not differ from the baseline regarding runtime improvements.

We see a tendency that the improvement percentage increases the more devices we test with. That is true for all architectures. For example, P-GNN scores 32.717% when the simulation works with 3 devices, 36.084% when working with 5 devices, and 37.114% with 8 devices.

	3 devices	**5 devices**	**8 devices**	**Overall average**
Placeto	24.585%	28.250%	34.339%	*29.058%*
GraphSAGE	29.566%	30.567%	31.374%	*30.502%*
P-GNN	32.717%	36.084%	37.114%	*35.305%*

Table 4.3: Summary of the achieved runtime improvements (in %) on the $cifar10$ dataset with 3, 5, and 8 devices.

nmt

Similar to our $cifar10$ experiment results, the results of the experiments we conduct on the nmt dataset further increase our certainty in our hypotheses. Namely, we see again that the P-GNN architecture yields the best runtime improvements out of all three tested architectures. The difference here is that

Placeto performs better than GraphSAGE, and we do not see the similarity in the results produced in the $cifar10$ experiments.

Our experiment on the nmt dataset with 3 devices show very close results between all three architectures (see Figure 4.6). P-GNN has the best results, with an average of 30.232% runtime improvements. However, the difference between the architecture with the second-best result, Placeto, is rather small - only about 2.125%. Placeto achieves an average runtime improvement of 28.107%. Conversely, GraphSAGE yields results that are quite close to Placeto - 27.037%.

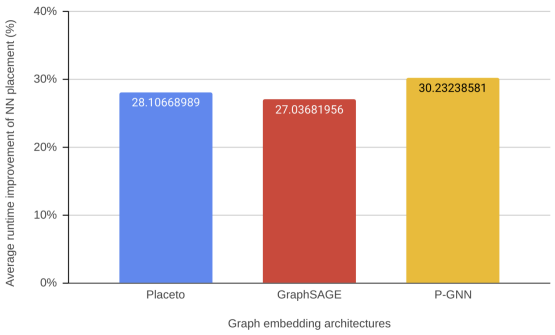


Figure 4.6: Average improvement of the runtime based on suggested placements in %. Experiment is run on the nmt dataset. Simulator classifies nodes on 3 devices.

When simulating 5 devices, we see similar results (see Figure 4.8). P-GNN takes the lead with 39.350%; Placeto is again second with 36.040%; GraphSAGE achieves 34.177%.

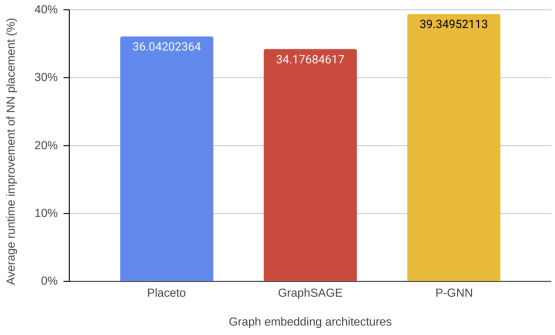


Figure 4.7: Average improvement of the runtime based on suggested placements in %. Experiment is run on the nmt dataset. Simulator classifies nodes on 5 devices.

Our simulation with 8 devices further solidifies the results when using the nmt dataset (see Figure 4.8). Namely, P-GNN achieves the best average runtime improvement - 39.350%. Placeto reaches 36.040%, while GraphSAGE yields 34.177%.

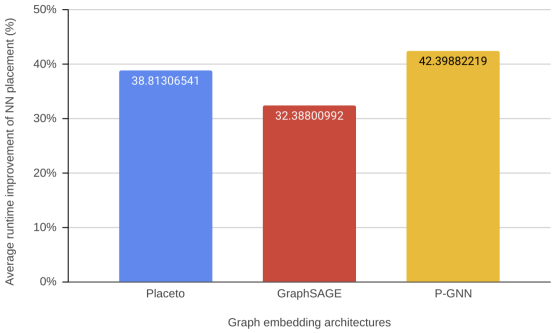


Figure 4.8: Average improvement of the runtime based on suggested placements in %. Experiment is run on the nmt dataset. Simulator classifies nodes on 8 devices.

In Table 4.4 we summarize our simulation results from the nmt dataset. P-GNN provides the biggest average runtime improvement - 37.327%. Placeto and GraphSAGE improve the runtime average compared to the initial placement, but by a smaller percentage - 34.321% and 31.201% respectively.

Once again, we see signs of the trend that the more devices are simulated, the more overall improvement the architectures can provide. As an example, P-GNN provides 30.232% for 3 devices, 39.350% for 5, and 42.399% for 8 devices. The same is true for Placeto. On the other hand, the GraphSAGE architecture does not follow the same path - 5 devices experiment yields better absolute results than the 8 devices one. However, the differences are small.

	3 devices	**5 devices**	**8 devices**	**Overall average**
Placeto	28.107%	36.042%	38.813%	*34.321%*
GraphSAGE	27.037%	34.177%	32.388%	*31.201%*
P-GNN	30.232%	39.350%	42.399%	*37.327%*

Table 4.4: Summary of runtime improvements on the nmt dataset in %

ptb

Running our experiments on the ptb dataset brings similar results in terms of which graph embedding architecture brings the most runtime improvements.

Similar to experiments conducted on nmt and $cifar10$, P-GNN achieves the best results. GraphSAGE and Placeto lack behind with GraphSAGE being the better of the two.

Simulating 3 devices shows a big gap between P-GNN (17.930%) and Placeto (8.981%), with GraphSAGE scoring improvements somewhere in between the former two (12.492%).

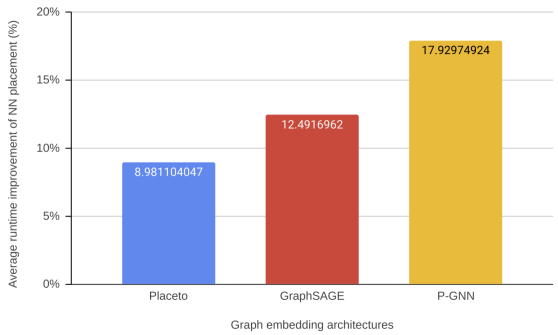


Figure 4.9: Average improvement of the runtime based on suggested placements in %. Experiment is run on the ptb dataset. Simulator classifies nodes on 3 devices.

Similar results are achieved when the experiment simulates 5 devices (Figure 4.10). Here, P-GNN reaches runtime improvement of 21.117%, while Placeto only improves the initial state with 11.304%. GraphSAGE (14.658%) again performs better than Placeto but is nowhere near the percentage achieved by P-GNN.

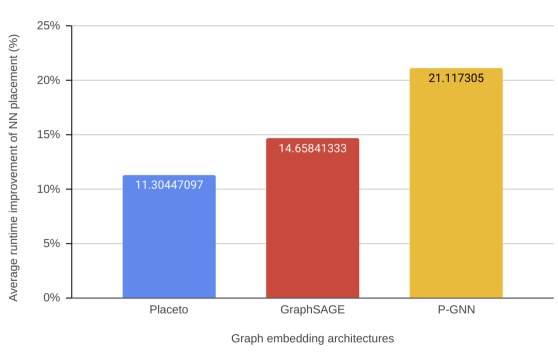


Figure 4.10: Average improvement of the runtime based on suggested placements in %. Experiment is run on the ptb dataset. Simulator classifies nodes on 5 devices.

In our last experiment, where we simulate 8 devices, the results are in line with the other experiments on the ptb dataset. P-GNN once again achieves the best runtime improvement (23.536%). GraphSAGE is second with almost 7% smaller improvement - 16.625%. Placeto yields the worst results - 15.831%. However, the difference between GraphSAGE and Placeto is not as pronounced as in the previous experiments.

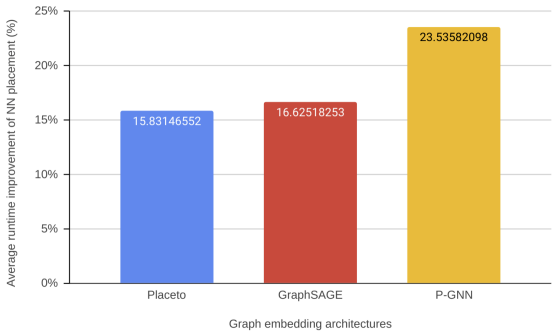


Figure 4.11: Average improvement of the runtime based on suggested placements in %. Experiment is run on the ptb dataset. Simulator classifies nodes on 8 devices.

In Table 4.5 we see a summary of our results on the ptb dataset. On average, Placeto improves the initial placement policy with 12.039%, GraphSAGE achieves 14.592%, and finally - P-GNN yields the biggest improvements of 20.861%.

There are a couple of things to note regarding the experiments on the ptb dataset.

First, similar to the experiments conducted on $cifar10$ and nmt, we see that all architectures improve the runtime progressively when more devices are added. As an example - P-GNN improves 17.930% the placement on 3 devices, 21.117% on 5, and 23.536% when 8 devices are simulated.

Second, the ptb dataset contains the largest input graphs. At the same time, the absolute improvements from all graph embedding architectures are smaller than the improvements on the other datasets. For example, GraphSAGE improves runtimes on $cifar10$ with 30.502% and nmt with 31.201%, and only 14.592% on ptb. The situation with Placeto and P-GNN is similar.

	3 devices	5 devices	8 devices	Overall average
Placeto	8.981%	11.304%	15.831%	*12.039%*
GraphSAGE	12.492%	14.658%	16.625%	*14.592%*
P-GNN	17.930%	21.117%	23.536%	*20.861%*

Table 4.5: Summary of runtime improvements on the ptb dataset in %

Summary

Showing the effects on runtime improvement of placement policies from different graph embedding architectures is one of the main goals of this thesis. We show that all of the tested architectures bring improvements to the runtime of the placement policy. Figure 4.12 serves as a confirmation of two of our hypotheses.

First, we see that the Placeto and GraphSAGE architectures achieve almost identical average improvements - 25.139% and 25.432% respectively. There are variations in the results between the different datasets. Nonetheless, the overall average is identical due to the similarity of the architectures. This result is in line with our expectations.

Second, we hypothesised that P-GNN would achieve more significant runtime improvements because it embeds more information about the input graph - particularly the locality of the input graph's nodes. P-GNN yields better results than the other two architectures, with an average of 31.164%. That is an improvement of more than 20% compared to GraphSAGE and Placeto and supports our hypothesis.

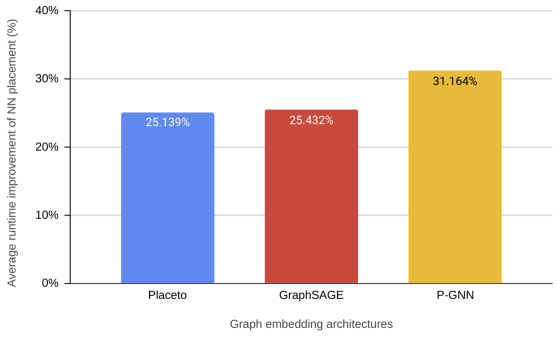


Figure 4.12: Average improvement of the runtime based on suggested placements in % from all experiments on all datasets.

Table 4.6 shows an aggregation of all the runtime improvements achieved

by each graph embedding architecture on each dataset. It also displays the summarized overall average runtime improvement for each architecture. We also outline the runtime improvement compared to our baseline - the default Placeto architecture. As mentioned, GraphSAGE provides similar runtime improvements, and it improves policy runtimes by only 1.165%. P-GNN has a significant lead compared to Placeto - 23.967%.

	cifar10	***nmt***	***ptb***	**Overall average**	**Improvement (%)**
Placeto	29.058%	34.321%	12.039%	*25.139%*	-
GraphSAGE	30.502%	31.201%	14.592%	*25.432%*	**+1.165%**
P-GNN	35.305%	37.327%	20.861%	*31.164%*	**+23.967%**

Table 4.6: Overall average runtime improvements for all graph embeddingarchitectures in %. Bigger improvement is better.

4.2.3 Computation time

In this section, we are going to show how the different graph embedding architectures impact the computation time needed by the RL agent. Similar to Section 4.2.2, this section is separated by datasets. Each subsection contains the results of all experiments conducted on the particular dataset. Last, we summarize all of the computation time results.

cifar10

In our experiments on the $cifar10$ dataset with 3 devices, the results consistently meet our expectations that the P-GNN architecture would require more time to complete than GraphSAGE. However, what we did not expect was for the default Placeto architecture to be the slowest of the three. As shown in Figure 4.13, Placeto has an average time of 110.730 seconds per episode for all input NNs. P-GNN averages the NN time of the RL agent at 101.170 seconds. GraphSAGE improves on Placeto by 19.519%, and on P-GNN with 11.915% with an average NN runtime of 89.116 seconds.

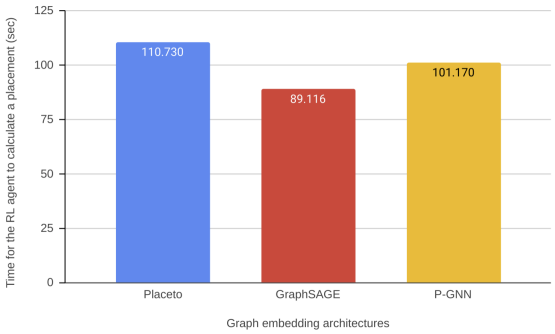


Figure 4.13: Average computation time for the RL agent in seconds per input NN. Experiment is run on the $cifar10$ dataset. Simulator classifies nodes on 3 devices.

When simulating 5 devices, the results from the 3 devices experiment are repeated almost entirely (Figure 4.14). The relation between the average times is kept the same. The only difference is the absolute increase in the average time, which is present for all architectures. More concretely, P-GNN is on average slower than GraphSAGE with 105.721 seconds compared to 91.930 seconds. Placeto once again requires the most computation time of all three architectures on average - 112.030 seconds.

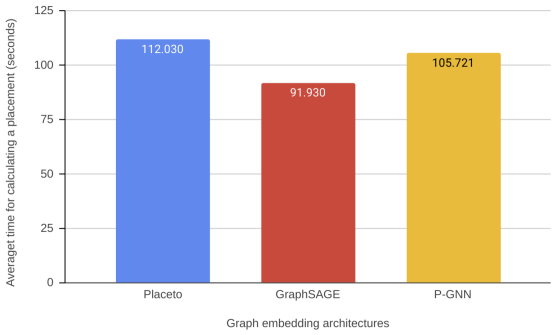


Figure 4.14: Average computation time for the RL agent in seconds per input NN. Experiment is run on the $cifar10$ dataset. Simulator classifies nodes on 5 devices.

Our last $cifar10$ experiment simulates 8 devices. The computation time continues the same tendency as with the previous two experiments (with 3 and 5 devices). GraphSAGE requires the least computation time with an average of 93.443 seconds. P-GNN requires 10 seconds more on average - 103.420

seconds. Placeto is again the slowest architecture, requiring 108.118, which is 15.705% more than GraphSAGE.

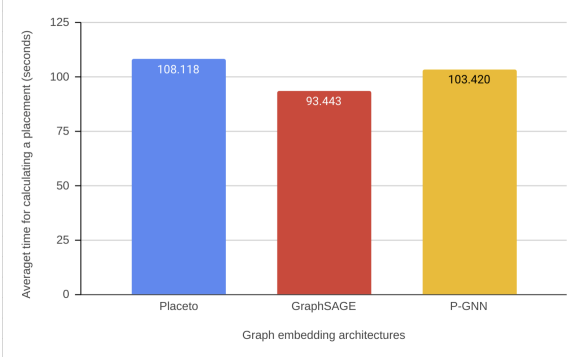


Figure 4.15: Average computation time for the RL agent in seconds per input NN. Experiment is run on the $cifar10$ dataset. Simulator classifies nodes on 8 devices.

Table 4.7 summarizes the average computation times for the three graph embedding architectures for each experiment. The overall averages show GraphSAGE as the fastest architecture on the $cifar10$ dataset with 91.497 seconds. P-GNN averages 103.437 seconds, while Placeto is the slowest with 110.293 seconds.

	3 devices	***5 devices***	***8 devices***	**Overall average**
Placeto	110.730	112.030	108.118	*110.293*
GraphSAGE	89.116	91.930	93.443	*91.497*
P-GNN	101.170	105.721	103.420	*103.437*

Table 4.7: Summary of average computation time on the $cifar10$ dataset in seconds

nmt

Experiments on the nmt dataset produce outcomes that are somewhat similar to the ones produced by the experiments on $cifar10$. The results demonstrate that GraphSAGE is the architecture that needs the least amount of computation time. We also observe that P-GNN and Placeto require similar computation times, with Placeto being marginally faster.

For the 3 devices experiment (Figure 4.16) GraphSAGE on average requires 42.504 seconds, whereas Placeto needs 47.054 seconds. Lastly - P-GNN is the slowest with 47.787 seconds.

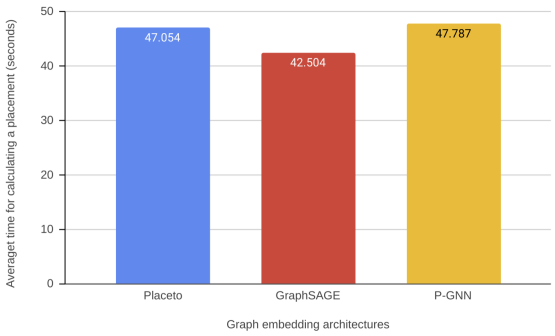


Figure 4.16: Average computation time for the RL agent in seconds per input NN. Experiment is run on the nmt dataset. Simulator classifies nodes on 3 devices.

The simulation with 5 devices shows similar results with GraphSAGE being the fastest, and P-GNN being the slowest architecture (Figure 4.17). GraphSAGE achieves 43.788 seconds, Placeto reaches 47.519 seconds, while P-GNN - 49.009 seconds on average.

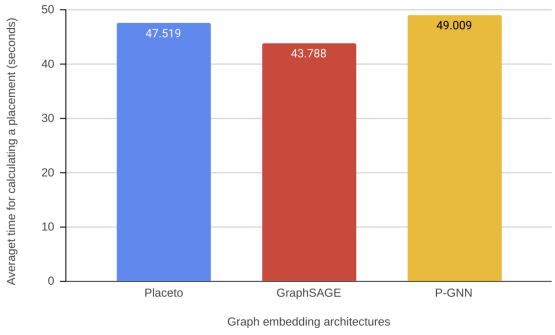


Figure 4.17: Average computation time for the RL agent in seconds per input NN. Experiment is run on the nmt dataset. Simulator classifies nodes on 5 devices.

Our experiment on nmt with 8 devices further confirms the previous result on that dataset (Figure 4.18). GraphSAGE scores 43.550 second, while Placeto is second with 48.996. P-GNN has the largest computation time, but the difference with Placeto is less than a second, namely 49.670 seconds.

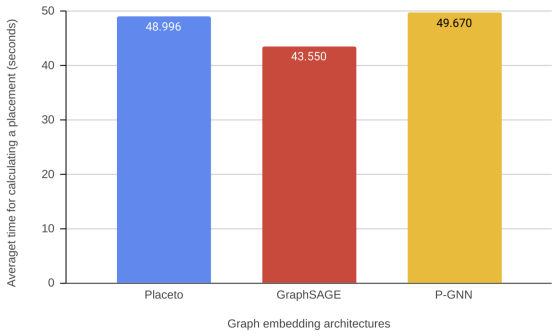


Figure 4.18: Average computation time for the RL agent in seconds per input NN. Experiment is run on the nmt dataset. Simulator classifies nodes on 8 devices.

Table 4.8 summarizes all computation times from all experiments conducted on the nmt dataset. The results are consistent in all experiments on nmt. More concretely, GraphSAGE is requiring less time than all the other architectures averaging at 43.281 seconds. Placeto and P-GNN are slower, with the former always outperforming the latter. Placeto averages 47.856 seconds over all experiments, whereas P-GNN achieves 48.822 seconds.

	3 devices	***5 devices***	***8 devices***	**Overall average**
Placeto	47.054	47.519	48.996	*47.856*
GraphSAGE	42.504	43.788	43.550	*43.281*
P-GNN	47.787	49.009	49.670	*48.822*

Table 4.8: Summary of average computation time on the nmt dataset in seconds

ptb

We conducted our last experiments on the ptb dataset. The results on this dataset affirm what we observe in our experiments with the other two datasets.

In the experiment with three devices, GraphSAGE requires the least computation time - 288.443 seconds. Placeto needs 335.376 seconds on average. P-GNN uses up the most computation time and averages 362.761 seconds (see Figure 4.19).

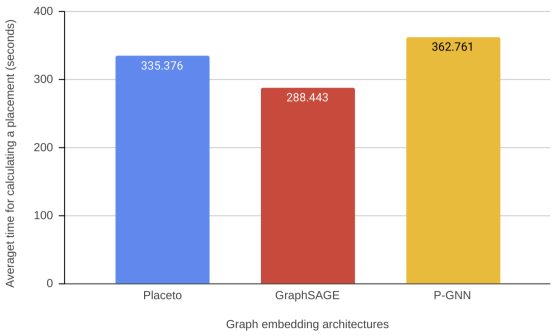


Figure 4.19: Average computation time for the RL agent in seconds per input NN. Experiment is run on the ptb dataset. Simulator classifies nodes on 3 devices.

Figure 4.20 shows the results of the experiments with 5 devices. We observe a similar pattern - GraphSAGE is quickest with 298.736 seconds, and P-GNN is the slowest at 368.460. Placeto's computation time is in between the other two - 327.175 seconds.

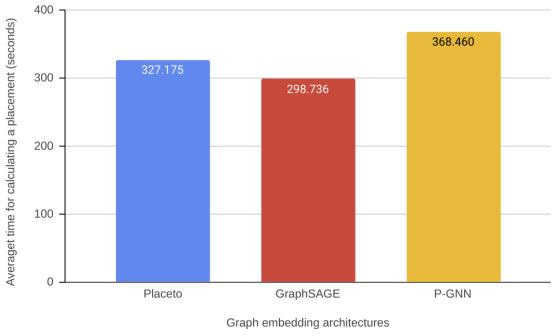


Figure 4.20: Average computation time for the RL agent in seconds per input NN. Experiment is run on the ptb dataset. Simulator classifies nodes on 5 devices.

Lastly, in Figure 4.21 we see the results from our experiments on ptb with 8 devices. The computation times are similar to the ones reached in the other experiments. GraphSAGE averages 298.736 seconds, Placeto - 327.175 seconds, and P-GNN - 368.460 seconds.

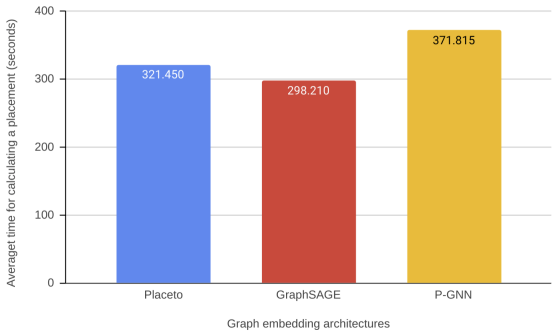


Figure 4.21: Average computation time for the RL agent in seconds per input NN. Experiment is run on the ptb dataset. Simulator classifies nodes on 8 devices.

This dataset contains the largest input graphs. That is the reason why the average computation times here (around 330 seconds) are so high compared to the computation times achieved using nmt (average of around 46 seconds) or even using $cifar10$ (average of around 101 seconds). The size, however, does not have an impact on how the architectures rank (see Table 4.9). GraphSAGE is 11.137% faster than Placeto and 24.582% faster than P-GNN. GraphSAGE has average computation time of 295.130 seconds, Placeto scores 328.000 seconds, and P-GNN achieves 367.679 seconds.

	3 devices	***5 devices***	***8 devices***	**Overall average**
Placeto	335.376	327.175	321.450	*328.000*
GraphSAGE	288.443	298.736	298.210	*295.130*
P-GNN	362.761	368.460	371.815	*367.679*

Table 4.9: Summary of average computation time on the ptb dataset in seconds

Summary

Our experiments provided data regarding the required computation time of the different architectures over different datasets. The results are homogeneous with regards to how the different graph embedding architectures are ranked in terms of their average computation times. Figure 4.22 visualizes that ranking. GraphSAGE shows the shortest computational times, with an average of 143.302 seconds. Placeto is second with 162.050 seconds. P-GNN is the slowest with an average of 173.313 seconds.

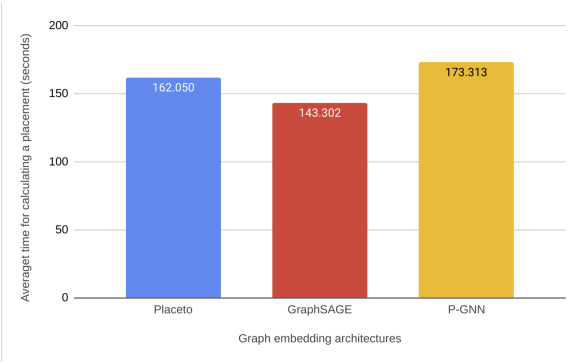


Figure 4.22: Cummulative average computation time for the RL agent in seconds per input NN

Provided that data, we can confirm our hypothesis that P-GNN requires the most time, as its complexity is more significant than the other two architectures. On the other hand, we do not have evidence to confirm our hypothesis that Placeto and GraphSAGE would require the same amount of computation time, as GraphSAGE is consistently faster.

It is noteworthy that the number of devices that the experiment uses does not definitively influence the computation requirements. In the runtime improvement experiments (Section 4.2.2), we observed a trend where the more devices the system simulated, the more significant the runtime improvement. The amount of devices does not affect the computation runtime for the experiments. For example, the average computation time of GraphSAGE is 42.504 seconds for 3 devices, 43.788 for 5 devices, and 43.550 seconds for 8 devices.

Table 4.10 summarizes the different computation times. It also shows the improvement percentage compared to our baseline, which is the default implementation - Placeto. GraphSAGE improves the computation time with 11.569%, while P-GNN adds an overhead of 6.950%.

	cifar10	***nmt***	***ptb***	**Overall average**	**Improvement (%)**
Placeto	110.293	47.856	328.000	*162.050*	-
GraphSAGE	91.497	43.281	295.130	*143.302*	***+11.569%***
P-GNN	103.437	48.822	367.679	*173.313*	*-6.950%*

Table 4.10: Overall average computation time for all graph embedding architectures per input NN in seconds. Bigger improvement is better.

4.2.4 P-C ratio

The P-C ratio is a metric that might give us a hint about the usefulness of a graph embedding architecture. It takes into account the runtime improvement and relates it to the computation time and is calculated as the ratio between the two. For example, if an architecture achieves 25% runtime improvement over 100 seconds, it would get the same P-C ratio as an architecture which brings less runtime improvement - 20% but quicker - 80 seconds, as $\frac{25\%}{100s} = \frac{20\%}{80s} = 0.25$.

Table 4.11 gives us an interesting insight. The average of P-C ratio of Placeto is 0.339, GraphSAGE scores 0.368, and P-GNN has the highest ratio of 0.387. To better understand, we look at Table 4.10 containing computation time summaries, as well as Table 4.6 containing the runtime improvement summaries. What we see is that despite being the slowest, P-GNN still brings the highest performance gain due to its large runtime improvements. On the other hand, GraphSAGE achieves similar runtime improvements with the baseline - Placeto. However, due to its comparatively fast speeds, it has a higher P-C ratio than Placeto.

	cifar10	***nmt***	***ptb***	**Average**	**Improvement (%)**
Placeto	0.264	0.716	0.037	0.339	-
GraphSAGE	0.333	0.720	0.049	0.368	**8.555%**
P-GNN	0.341	0.763	0.057	0.387	**14.159%**

Table 4.11: P-C ratio summary in %/seconds

Chapter 5

Discussion and Conclusion

In this thesis, we have explored what is the impact of graph embeddings on the performance of device placement systems and the placement policies produced by them. The latest efforts in the field harness RL to automatically create device placement policies. Our work steps on one such example of state-of-the-art solutions - Placeto (Addanki et al. (2019)).

An essential part of the RL systems, including Placeto, is the graph embedding module. Our goal is to gauge the quality of placement policies created by Placeto using different graph embedding methods. We evaluate our work in two main areas. First, we evaluate runtime improvements - estimated improvement of the runtime of an input NN when using a policy. Second, we assess computation time - the time required to reach the achieved runtime improvement.

In order to assess how graph embeddings impact the performance of the placement policies, we replace its graph embedding module with implementations of current state-of-the-art graph embedding architectures, namely GraphSAGE (Hamilton, Ying, and Leskovec (2018)), and P-GNN (You, Ying, and Leskovec (2019)).

We expected that different graph embeddings would affect the quality of the placement policies differently. In Table 5.1, we present a summary of all the results we achieved in our experiments. We have looked at all of the dimensions of our experiments and their results separately and into much detail. Here, we will not go into such depths. Instead, we take a look at the whole picture of how graph embeddings influence the device placement policies.

First, we take a look at Placeto (Addanki et al. (2019)). Placeto, with its default graph embedding module, provides an improvement of 25.139%. The computation time required is 162.050 seconds. The ratio between the runtime

	Runtime improvements (%)		**Computation times (seconds)**		**P-C ratio (%/seconds)**	
	Average	Improvement (%)	Average	Improvement (%)	Average	Improvement (%)
Placeto	25.139	-	162.050	-	0.339	-
GraphSAGE	25.432	**1.165%**	143.302	**11.569%**	0.368	**8.555%**
P-GNN	31.164	**23.967%**	173.313	-6.950%	0.387	**14.159%**

Table 5.1: Complete summary of all results for all tested parameters.

improvement and the computation time (P-C ratio) is 0.339. The improvements achieved by this architecture are explored in the original paper. We use it as a baseline and compare all our results to it.

Second, GraphSAGE (Hamilton, Ying, and Leskovec (2018)), provides runtime improvements very similar to the ones of Placeto - 25.432%. The result is in line with our expectations, as the graph embedding architectures are somewhat similar to each other. What we did not expect is the computation time required by GraphSAGE. It reaches its runtime improvements for 143.302 seconds on average, which is 11.569% faster than Placeto. The larger speed helper GraphSAGE achieve a P-C ratio of 0.368.

Last, we take a look at our implementatin of P-GNN (You, Ying, and Leskovec (2019)). Since it has the most complex graph embedding architecture, we expected that it would bring the most significant runtime improvements. Indeed, our expectations are met, and P-GNN achieves 31.164% runtime improvement, which is 23.967% more than Placeto. As predicted, the improvement comes at the cost of slower computation times. P-GNN achieves its runtime results for an average of 173.313 seconds, which is 6.950% slower than the baseline. Regardless, the P-C ratio is still the highest of the three, as the substantial runtime improvement offsets the somewhat slower computation time.

Provided the results we achieved in our work, we can confirm that graph embeddings indeed have an impact on device placement policies. Furthermore, the more information about a graph is contained within a graph embedding, the better the runtime improvements. At the same time, the added complexity can harm the time required for the device placement system to achieve that policy.

Bibliography

[1] John Sunda Hsia. *Artificial Intelligence – A History*. en. Library Catalog: developer.att.com. 2018. URL: https://developer.att.com/blog/the-evolution-of-ai (visited on 06/11/2020).

[2] The Economist. *After Moore's law | Technology Quarterly*. Library Catalog: www.economist.com. Dec. 2016. URL: https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law (visited on 06/11/2020).

[3] Benjamin Recht et al. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-Taylor et al. Curran Associates, Inc., 2011, pp. 693–701. URL: http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf (visited on 06/11/2020).

[4] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1223–1231. URL: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf (visited on 06/11/2020).

[5] Hao Li et al. "MALT: distributed data-parallelism for existing ML applications". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, Apr. 2015, pp. 1–16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741965. URL: https://doi.org/10.1145/2741948.2741965 (visited on 06/11/2020).

[6] Henggang Cui et al. "GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, Apr. 2016, pp. 1–

16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901323. URL: https://doi.org/10.1145/2901318.2901323 (visited on 06/11/2020).

[7] Alexandros Koliousis et al. "CROSSBOW: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers". In: *arXiv:1901.02244 [cs]* (Jan. 2019). arXiv: 1901.02244. URL: http://arxiv.org/abs/1901.02244 (visited on 06/11/2020).

[8] Azalia Mirhoseini, Hieu Pham, et al. "Device Placement Optimization with Reinforcement Learning". In: *arXiv:1706.04972 [cs]* (June 2017). arXiv: 1706.04972. URL: http://arxiv.org/abs/1706.04972 (visited on 02/07/2020).

[9] Azalia Mirhoseini, Anna Goldie, et al. "A HIERARCHICAL MODEL FOR DEVICE PLACEMENT". en. In: (2018), p. 11.

[10] Ravichandra Addanki et al. "Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning". In: *arXiv:1906.08879 [cs, stat]* (June 2019). arXiv: 1906.08879. URL: http://arxiv.org/abs/1906.08879 (visited on 02/07/2020).

[11] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *arXiv:1706.02216 [cs, stat]* (Sept. 2018). arXiv: 1706.02216. URL: http://arxiv.org/abs/1706.02216 (visited on 03/17/2020).

[12] Jiaxuan You, Rex Ying, and Jure Leskovec. "Position-aware Graph Neural Networks". In: *arXiv:1906.04817 [cs, stat]* (June 2019). arXiv: 1906.04817. URL: http://arxiv.org/abs/1906.04817 (visited on 03/19/2020).

[13] Jack R. Meredith et al. "Alternative research paradigms in operations". en. In: *Journal of Operations Management* 8.4 (Oct. 1989), pp. 297–326. ISSN: 0272-6963. DOI: 10.1016/0272-6963(89)90033-8. URL: http://www.sciencedirect.com/science/article/pii/0272696389900338 (visited on 05/24/2020).

[14] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". en. In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (visited on 03/16/2020).

[15] Frank Rosenblatt. *The Perceptron: A Perceiving and Recognizing Automaton*. Tech. rep. Report 85–460–1. Cornell Aeronautical Laboratory, Jan. 1957.

[16] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. English. O'Reilly Media, Inc., Mar. 2017. ISBN: 978-1-4919-6228-2.

[17] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. en. Google-Books-ID: uyV5AgAAQBAJ. Psychology Press, 1949. ISBN: 978-1-135-63191-8.

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Internal Representations by Error Propagation*. en. Tech. rep. ICS-8506. CALIFORNIA UNIV SAN DIEGO LA JOLLA INST FOR COGNITIVE SCIENCE, Sept. 1985. URL: https://apps.dtic.mil/docs/citations/ADA164453 (visited on 03/16/2020).

[19] Ruben Mayer and Hans-Arno Jacobsen. "Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques and Tools". In: *arXiv:1903.11314 [cs]* (Sept. 2019). arXiv: 1903.11314. URL: http://arxiv.org/abs/1903.11314 (visited on 03/13/2020).

[20] Xiandong. *Intro Distributed Deep Learning*. May 2017. URL: xiandong79.github.io//intro-distributed-deep-learning (visited on 03/14/2020).

[21] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *arXiv:1404.5997 [cs]* (Apr. 2014). arXiv: 1404.5997. URL: http://arxiv.org/abs/1404.5997 (visited on 03/14/2020).

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond Data and Model Parallelism for Deep Neural Networks". In: *arXiv:1807.05358 [cs]* (July 2018). arXiv: 1807.05358. URL: http://arxiv.org/abs/1807.05358 (visited on 03/14/2020).

[23] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *arXiv:1603.04467 [cs]* (Mar. 2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467 (visited on 06/02/2020).

[24] Huasha Zhao and John Canny. "Sparse Allreduce: Efficient Scalable Communication for Power-Law Data". In: (Dec. 2013).

[25] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv:1802.05799 [cs, stat]* (Feb. 2018). arXiv: 1802.05799. URL: http://arxiv.org/abs/1802.05799 (visited on 06/12/2020).

[26] Thiago S. Guzella and Walmir M. Caminhas. "A review of machine learning approaches to Spam filtering". en. In: *Expert Systems with Applications* 36.7 (Sept. 2009), pp. 10206–10222. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2009.02.037. URL: http://www.sciencedirect.com/science/article/pii/S095741740900181X (visited on 06/12/2020).

[27] Tom Young et al. "Recent Trends in Deep Learning Based Natural Language Processing [Review Article]". In: *IEEE Computational Intelligence Magazine* 13.3 (Aug. 2018). Conference Name: IEEE Computational Intelligence Magazine, pp. 55–75. ISSN: 1556-6048. DOI: 10.1109/MCI.2018.2840738.

[28] John O. Awoyemi, Adebayo O. Adetunmbi, and Samuel A. Oluwadare. "Credit card fraud detection using machine learning techniques: A comparative analysis". In: *2017 International Conference on Computing Networking and Informatics (ICCNI)*. Oct. 2017, pp. 1–9. DOI: 10.1109/ICCNI.2017.8123782.

[29] Mahmoud M. Abu Ghosh and Ashraf Y. Maghari. "A Comparative Study on Handwriting Digit Recognition Using Neural Networks". In: *2017 International Conference on Promising Electronic Technologies (ICPET)*. Oct. 2017, pp. 77–81. DOI: 10.1109/ICPET.2017.20.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018. ISBN: 978-0-262-03924-6.

[31] *Part 2: Kinds of RL Algorithms — Spinning Up documentation*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below (visited on 05/24/2020).

[32] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". en. In: *Machine Learning* 8.3 (May 1992), pp. 229–256. ISSN: 1573-0565. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696 (visited on 06/16/2020).

[33] Lilian Weng. *Policy Gradient Algorithms*. en. Library Catalog: lilianweng.github.io. Apr. 2018. URL: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html (visited on 06/16/2020).

[34] *Definition of graph*. en. Library Catalog: www.merriam-webster.com. 2020. URL: https://www.merriam-webster.com/dictionary/graph (visited on 03/16/2020).

[35] *Graph theory*. en. Page Version ID: 945009584. Mar. 2020. URL: https://en.wikipedia.org/w/index.php?title=Graph_theory&oldid=945009584 (visited on 03/16/2020).

[36] Smriti Bhagat, Graham Cormode, and S. Muthukrishnan. "Node Classification in Social Networks". In: *arXiv:1101.3291 [physics]* (2011). arXiv: 1101.3291, pp. 115–148. DOI: 10.1007/978-1-4419-8462-3_5. URL: http://arxiv.org/abs/1101.3291 (visited on 06/12/2020).

[37] David Liben-Nowell and Jon Kleinberg. "The link-prediction problem for social networks". en. In: *Journal of the American Society for Information Science and Technology* 58.7 (2007). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.10 pp. 1019–1031. ISSN: 1532-2890. DOI: 10.1002/asi.20591. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.20591 (visited on 06/12/2020).

[38] Scott White and Padhraic Smyth. "A Spectral Clustering Approach To Finding Communities in Graphs". In: *Proceedings of the 2005 SIAM International Conference on Data Mining*. Proceedings. Society for Industrial and Applied Mathematics, Apr. 2005, pp. 274–285. ISBN: 978-0-89871-593-4. DOI: 10.1137/1.9781611972757.25. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972757.25 (visited on 06/12/2020).

[39] Tomás Feder and Rajeev Motwani. "Clique partitions, graph compression and speeding-up algorithms". In: *Proceedings of the twenty-third annual ACM symposium on Theory of Computing*. STOC '91. New Orleans, Louisiana, USA: Association for Computing Machinery, Jan. 1991, pp. 123–133. ISBN: 978-0-89791-397-3. DOI: 10.1145/103418.103424. URL: https://doi.org/10.1145/103418.103424 (visited on 06/12/2020).

[40] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. "Graph summarization with bounded error". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, June 2008, pp. 419–432. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376661. URL: https://doi.org/10.1145/1376616.1376661 (visited on 06/12/2020).

[41] Palash Goyal and Emilio Ferrara. "Graph Embedding Techniques, Applications, and Performance: A Survey". In: *Knowledge-Based Systems* 151 (July 2018). arXiv: 1705.02801, pp. 78–94. ISSN: 09507051. DOI: 10.1016/j.knosys.2018.03.022. URL: http://arxiv.org/abs/1705.02801 (visited on 03/17/2020).

[42] S. T. Roweis and L. K. Saul. "Nonlinear dimensionality reduction by locally linear embedding". eng. In: *Science (New York, N.Y.)* 290.5500 (Dec. 2000), pp. 2323–2326. ISSN: 0036-8075. DOI: 10.1126/science.290.5500.2323.

[43] Mikhail Belkin and Partha Niyogi. "Laplacian eigenmaps and spectral techniques for embedding and clustering". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*. NIPS'01. Vancouver, British Columbia, Canada: MIT Press, Jan. 2001, pp. 585–591. (Visited on 06/12/2020).

[44] Mingdong Ou et al. "Asymmetric Transitivity Preserving Graph Embedding". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, Aug. 2016, pp. 1105–1114. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939751. URL: https://doi.org/10.1145/2939672.2939751 (visited on 06/12/2020).

[45] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "DeepWalk: Online Learning of Social Representations". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14* (2014). arXiv: 1403.6652, pp. 701–710. DOI: 10.1145/2623330.2623732. URL: http://arxiv.org/abs/1403.6652 (visited on 06/12/2020).

[46] Aditya Grover and Jure Leskovec. "node2vec: Scalable Feature Learning for Networks". In: *arXiv:1607.00653 [cs, stat]* (July 2016). arXiv:

1607.00653. URL: http://arxiv.org/abs/1607.00653 (visited on 06/12/2020).

[47] Daixin Wang, Peng Cui, and Wenwu Zhu. "Structural Deep Network Embedding". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, Aug. 2016, pp. 1225–1234. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939753. URL: https://doi.org/10.1145/2939672.2939753 (visited on 06/12/2020).

[48] Shaosheng Cao, Wei Lu, and Qiongkai Xu. "Deep neural networks for learning graph representations". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, Feb. 2016, pp. 1145–1152. (Visited on 06/12/2020).

[49] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *arXiv:1609.02907 [cs, stat]* (Feb. 2017). arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907 (visited on 06/12/2020).

[50] Yuanxiang Gao, Li Chen, and Baochun Li. "Spotlight: Optimizing Device Placement for Training Deep Neural Networks". en. In: *International Conference on Machine Learning*. July 2018, pp. 1676–1684. URL: http://proceedings.mlr.press/v80/gao18a.html (visited on 02/09/2020).

[51] Azade Nazi et al. "GAP: Generalizable Approximate Graph Partitioning Framework". In: *arXiv:1903.00614 [cs, stat]* (Mar. 2019). arXiv: 1903.00614. URL: http://arxiv.org/abs/1903.00614 (visited on 02/07/2020).

[52] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks: Design and Applications*. 1st. USA: CRC Press, Inc., 1999. ISBN: 978-0-8493-7181-3.

[53] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997). Publisher: MIT Press, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (visited on 06/12/2020).

[54] Ravichandra Addanki. *Reference code for Placeto*. original-date: 2019-10-24T23:58:16Z. Mar. 2020. URL: https://github.com/aravic/generalizable-device-placement (visited on 06/02/2020).

[55] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. en. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. ISSN: 1938-7228 Section: Machine Learning. Mar. 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html (visited on 06/02/2020).

[56] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *arXiv:1912.01703 [cs, stat]* (Dec. 2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703 (visited on 06/08/2020).

[57] Hieu Pham et al. “Efficient Neural Architecture Search via Parameter Sharing”. In: *arXiv:1802.03268 [cs, stat]* (Feb. 2018). arXiv: 1802.03268. URL: http://arxiv.org/abs/1802.03268 (visited on 06/10/2020).

[58] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. en. In: (2009), p. 60.

[59] Mitchell P. Marcus et al. *Treebank-3*. [language = eng] English. [linguistic-type = primary_text]. ISBN: 9781585631636 Library Catalog: olac.ldc.upenn.edu Publisher: Linguistic Data Consortium. 1999. URL: http://olac.ldc.upenn.edu/item/oai:www.ldc.upenn.edu:LDC99T42 (visited on 06/10/2020).

[60] Tensorflow. *Neural machine translation with attention | TensorFlow Core*. en. Library Catalog: www.tensorflow.org. URL: https://www.tensorflow.org/tutorials/text/nmt_with_attention (visited on 06/10/2020).

[61] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. In: *arXiv:1508.04025 [cs]* (Sept. 2015). arXiv: 1508.04025 version: 5. URL: http://arxiv.org/abs/1508.04025 (visited on 06/10/2020).