Graph neural networks for robotic manipulation

Fabio d'Aquino Hilt [1] **JanNiklas Kolf** [1] **Christian Weiland** [1] **João Carvalho** [2]

Abstract

Graph neural networks (GNNs) are increasingly used for task planning in robot manipulation, because they work better with the inherent graph structure of manipulation problems. We designed a task planning problem with arbitrary numbers of objects similar to reorganizing a pantry (in-place reorganizing of stacked objects) and trained both a GNN and a feed-forward neural network (NN) to solve it. The GNN consistently generalizes faster and achieves higher accuracy ($> 90\%$) than the NN and when trained with 5 objects per scene achieves zero-shot generalization with $> 85\%$ accuracy on unseen datasets with 5 to 15 objects in the scenes.

1. Introduction

Task planning is a crucial part of robotics and solving this problem has been of increased popularity recently. With deep learning new possibilities in this topic arrived. Graph neural networks (GNNs) are one specific type of neural network that work natively in graph domains. Using graphs to represent the objects in a scene and the relations between them has proven to be a convenient representation(Scarselli et al., 2009) and further cemented the usefulness of GNNs for task planning. While traditional neural networks cannot generally handle dynamic inputs due to the fixed network structure and the fixed size of the weight matrices, the graph-shaped inputs of GNNs can have any shape or size since GNNs process every node independently and only the feature vectors of every node must have the same size. This makes it possible to use a single GNNs where previously several different networks would have to be trained for every different input size. Chapter 3.1 presents the functionality of GNNs in detail. In this paper we design a simple task planning problem and train both a feed forward neural network (NN) and a GNN to solve it. We show that the GNN seems to generalize both better and faster than the NN and generalizes well to bigger input graphs (with more nodes) than the graphs on which it was trained. We also set up simulated vision and an action components which in future can be integrated with the task planning network to create a fully connected simulated robotic manipulation pipeline and further confirm the applicability of GNNs to task planning in robotic manipulation problems.

[1]Technical University of Darmstadt, Darmstadt, Hesse, Germany [2]Intelligent Autonomous Systems Lab, Technical University of Darmstadt, Darmstadt, Hesse, Germany. Correspondence to: Fabio d'Aquino Hilt <fabio.daquinohilt@stud.tu-darmstadt.de>, Jan Niklas Kolf <janniklas.kolf@stud.tu-darmstadt.de>, Christian Weiland <christian.weiland@stud.tu-darmstadt.de>, João Carvalho <joao.correia_carvalho@tu-darmstadt.de>.

2. Related Work

Task and motion planning (TAMP) is a fundamental part of robotics. To perform a trajectory, knowledge about the environment and the task is required (Ren et al., 2021). The complexity of the scene and the length of the tasks and executions make TAMP significantly more difficult. Many different solutions have been presented for the problem (Ren et al., 2021).

Ren et al. have a symbolic approach in their work (Ren et al., 2021) using symbolic top-k planning. The presented eTAMP algorithm outperformed the compared algorithms in performance and speed in the empirical tests.

The problem of task planning can be made more difficult by the fact that too many objects have to be considered during planning, which rapidly reduces the performance of the calculation. In order to ensure an efficient calculation, Silver et al. (Silver et al., 2020) presented a system that extracts the relevant objects for planning from the set of all existing objects with the help of graph neural networks. For this purpose, they introduced the notation of object importance scores, which predicts how relevant an object is for the task planning calculation. The authors determined the respective labels for the calculations through an approximation in the test environment. These scores are then used in a supervised learning approach to learn a graph neural network model. In various test scenarios, the authors were able to show that their method significantly improved the calculation time and that the approach beats other approaches in the tests.

To create actions for solving TAMPs, Driess et al. (Driess et al., 2020) use a photo of a scene, which together with actions and objects is fed into a recurrent neural network with its own encoder, which has a sequence of actions as a result. This can eliminate the time-consuming search for

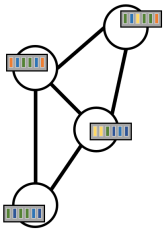

Figure 1. An example graph with 4 nodes. Each node has a six dimensional feature vector and is connected to neighboring nodes. The edges have no weights.

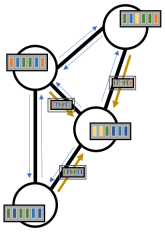

Figure 2. Each node sends its own feature vector to the neighboring nodes in the message passing step. Highlighted are the messages that reach the middle right node. You can see how the neighboring nodes send their own feature vectors to this node.

sequences, as well as the need to constantly observe the scene. In their experiments, the authors were able to show that even with only two objects during training, the network could also generalize to multiple objects.
Lin et al. (Lin et al., 2021) also achieved a generalization by training a GNN based on expert-trajectories. The system, which represented start and target positions as nodes in the graph. Graph convolutions are used to select an object that is to be moved by a pick and place action. Their system also generalizes to multiple objects, where the number of objects was not used during training.

3. Preliminaries

In this section we give a short introduction to GNNs and describe the building blocks of a robotic manipulation pipeline.

3.1. Graph neural networks - GNNs

In many application areas, such as image analysis, scene description, software engineering, and natural language processing, data naturally has a graph structure or a non-euclidean structure which can easily be represented by a graph (Scarselli et al., 2009). A graph consists of different nodes $\mathbf{N}$, as well as edges $\mathbf{E}$, which connect the nodes respectively. Also, each node n_i has a feature vector $\mathbf{x}_i$ of the same dimension m. Edges can be assigned different weights, which can also be understood as costs for actions between the nodes. The nodes do not have to be connected in pairs, but the graph and the connections between the nodes can be defined freely. The set of neighboring nodes of node i is denoted as $\mathcal{N}_i$. An example graph is shown in figure 1.

Depending on the use case, the structure of a GNN may differ. In this paper, we consider the case where for each node n_i the initial feature vector $\mathbf{x}_i$ of dimension m is transformed to a target vector $\tilde{\mathbf{x}}_i$ of a chosen dimension $\tilde{m}$. This can also be called node prediction, since a feature vector is to be predicted for each node. The construction of the features is carried out according to a fixed principle. As a first step, the individual nodes send their current feature vector to all nodes to which it is connected via edge. The nodes each collect all incoming messages from their direct neighbors. This step is called message passing. In figure 2 the message passing for a node is illustrated. Once all messages with feature vectors have been received, they are put into a so-called convolution. This is similar to a convolution layer in a convolutional neural network, a function that combines the current feature vector with the incoming feature vectors. By linear or also non-linear transformations and multiplications with weight matrices features of different dimension can be achieved. As an example, since it is also used in this work, the convolution GraphConv of Weisfeiler et al. (Morris et al., 2019) is presented. A new feature $\mathbf{x}'_i$ for node i is computed with

$$\mathbf{x}'_i = \mathbf{\Theta}_1 \mathbf{x}_i + \mathbf{\Theta}_2 \sum_{j \in \mathcal{N}(i)} e_{j,i} \cdot \mathbf{x}_j$$

where $\mathbf{\Theta}_1$ and $\mathbf{\Theta}_2$ are weight matrices with respective size (m', m). This maps a feature vector of dimension m to dimension m' with a linear matrix multiplication. All received feature vectors of neighbors j are multiplied by the edge weight e_{jni} between the current node i and the sending node j, and the individual features are summed element by element. Then, the summed feature vector is multiplied by the weight matrix $\mathbf{\Theta}_2$ and the two transformed vectors are added to form the final, updated feature vector $\mathbf{x}'_i$. Similar to neural networks, the computed feature vectors can be given by linear or non-linear activation functions as well as further regularization methods like batch normalization or dropout can be applied. Figure 3 shows how a node puts the entire feature vectors into a function and generates the new feature vector from it.

By applying the message passing, convolutions and activation steps multiple times, feature vectors can also be propagated and aggregated over multiple node hops, with enough convolutions even over very long distances. Likewise, known loss functions from the NN domain can be applied to the feature vectors and learning the individual parameters also works by backpropagation.

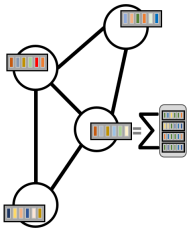

Figure 3. The node has collected all incoming feature vectors and now applies an arbitrary function Σ to feature vectors, resulting in a new feature vector.

3.2. Robotic Manipulation Pipeline

A full robotic manipulation pipeline needs vision to recognize objects in images and discover their location and orientation, task planning to decide how to manipulate which object, and a simulated environment and robot arm with a pick-and-place algorithm to manipulate the chosen object according to the planned task. Here we describe a simulated pipeline, but in a real world pipeline the components are similar, just the simulation is replaced by real objects, a camera, and a robot arm.

Vision - The first part of the pipeline consists of detection of objects and their positions. In a simulated environment the image of the emulated camera is used to detect the positions and orientations of the objects in the scene. Tremblay et al. have shown (Tremblay et al., 2018) how deep learning can be used to predict the position and orientation of YCB objects (Calli et al., 2017). The 6-degree-of-freedom prediction enables systems to give robots more precise grasping instructions and place objects more accurately. The network architecture used is a one-short fully convolutional neural network with a multistage architecture. It produces a belief map and a vector field for individual vertices of the bounding box. From the created features, the system extracts a bounding box that encloses the object, from which the position and orientation of the object is calculated.
To determine the positions of the provided image by the environment, the pretrained neural network by Tremblay et al. is used (Tremblay et al., 2018).

Task planning - The second part of a pipeline is planning and deciding which object should be placed where. Such operations are called "task planning".
A fully observable, deterministic task planning problem is described by the formal language PDDL (Fox & Long, 2003) as a tuple $\langle A, s_0, g \rangle$, where A is a set of actions defined by preconditions and effects, s_0 is an initial state of the domain and g is a set of goal conditions. This means a sequence of actions consists of actions $a_0, \ldots, a_n$, and a sequence of states consists of $s_0, \ldots, s_{n+1}$. A state transition follows the common term $s_{t+1} = f(s_t, a_t)$, where the next state s_{t+1} is reached by performing action a_t in state s_t, following the state transition function f. The solution to the planning problem is an action sequence where s_t satisfies the preconditions of a_t for $t = 0, \ldots, n$ and s_{n+1} satisfies the goal state g. The task planner can also be used in an iterative fashion to find not an action sequence, but only the next action a_t for the current state s_t. Executing the task planner and the state transition function in a loop results in the same action sequence as before, but with a fixed size output of the task planner. This approach is especially interesting for neural networks which have fixed output sizes.

Simulation of environment and robot arm - Simulations are the go-to way for experiments and testing of new methods and models. They ensure a controlled environment in which no hardware can be destroyed and allow for quicker development speed since the environment is simpler and easier to control, and potential safety breaches can be excluded at first. At the same time, a simulation is not a completely accurate representation of the real world and therefore a transformation of the model and application to the real world brings new problems and errors. An example simulation framework is CoppeliaSim (CoppeliaRobotics), which can be combined with models and control software for simulated robot arms, like ikida (IAS) for a panda robotic arm with a Franka gripper (ORI). Simulation environments are often well integrated with the Robot Operating System (ROS), which is used in real robots. This helps to deploy a working model timely to a real robot.

4. Investigations

In this section, the process as well as the details of implementation, data generation, and the structure of the GNN as well as the NN are presented, and key research questions are considered and answered.

4.1. Experimental Setup

Environment - The environment is assumed to be a flat surface with defined width and depth on which various objects are stacked. The goal is to develop scenarios that can be simulated with the YCB data set. A typical household task is to move objects and place them at specific target positions. The task planner must therefore be able to determine when and how objects can or cannot be moved. An object is movable if no other object is standing on it and it is not completely surrounded by other objects.Therefore, it is not movable if other objects are standing on it or if the object is otherwise obscured. However, it is not always useful to

move an arbitrary free object, but preferably a free object, which can be moved to the target position in one move. Depending on the scenario, this can be a very challenging task for the task planner.

Data generation - The data as well as the individual data sets consist of scenes. All scenes of a dataset consist of a fixed, given number of objects. For most of the experiments, the number of objects was arbitrarily fixed at $n = 5$, which seemed a reasonably small number where objects already start to stack on top of each other. Also the dataset gets possible object sizes and the size of the workspace given.
A scene is created by sampling a starting position from the workspace for each object, which is either on an already placed object or where the object to be placed has no overlap with other objects. This procedure is also repeated for the target positions. Start and target positions are therefore independent of each other.
Through an algorithm that takes into account the current positions and the target positions, the labels are created for all objects in the scene. The target labels for each object are ⟨ Move object to target⟩ , ⟨Leave object standing⟩ and ⟨Move object to intermediate position⟩. The features of the respective objects in a scene consist of the three-dimensional start position $\langle x, y, z\rangle_{start}$ as well as the three-dimensional target position $\langle x, y, z\rangle_{goal}$ and therefore form a six-dimensional feature vector. An object forms a node in the graph where each node is connected to all other nodes in the graph, forming a fully connected graph.
The next scene is generated by moving an object from the current scene. If possible, an object is moved to its target position and if this is not possible, an object is moved to a free intermediate position. This iterative procedure is repeated until all objects are on their target positions. If this is the case, a completely new scene with new start and target positions is generated and solved again by the iterative procedure. If an object is supposed to be moved, left standing or moved to an intermediate position in a step of the solution procedure, the respective value is set to 1 in the three-dimensional ground truth vector while the remaining values remain at 0. In total, about 1000 scenes were created for one data set. A created example of how a scene could look like can be seen in figure 4.

Workflow details - For the training of the networks, only one created data set with 1000 scene graphs with 5 objects each was used. The graphs were randomly shuffled and 70% of the data was used for training, 20% fot testing, and 10% for validation. For final evaluation on datasets, 1000 scenes were again created and the performance metrics were evaluated on these entirely new data points. These data points were not used for training and the network or system had not seen these scenes in advance. Both neural networks were trained with the same hyperparameters which work reasonably well for both.

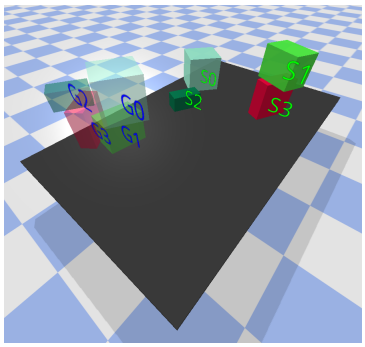

Figure 4. Example scene setup. The initial positions of the objects are marked with SX while the goal positions of the respective objects are labeled as GX. The objects and their positions were created randomly.

Graph neural networks - The Graph neural network uses as convolution two GraphConv layers, each followed by a ReLU, in a row, the first with a matrix size of $\langle 6, 50\rangle$ and the second GraphConv uses a weight matrix size of of $\langle 50, 50\rangle$. This is followed by a Linear layer with a layer size of $\langle 50, 3\rangle$. The network uses cross-entropy loss as a loss function. The prediction label is the class that has the largest value within the three-dimensional output. A learning rate of $3e{-}3$ was chosen with Adam as the optimizer.

Neural networks - Since the neural network requires a fixed input size, the six dimensional features of each of the five objects were concatenated into a vector. Therefore, the NN has an input size of 30. The network has a similar structure to the GNN with Linear$\langle 30, 50\rangle$, ReLU, Linear$\langle 50, 50\rangle$, ReLU, Linear$\langle 50, 15\rangle$. Cross-entropy loss was again used as the loss function, with the target vectors also concatenated to match the fifteen dimensional output vector. As learning rate $3e{-}3$ gave the best results still with Adam as the optimizer.

Investigations - We divide our investigations into three parts:

(a) We show that GNNs have significantly better performance than NNs in both achieved accuracy and learning efficiency.

(b) We show that the GNN is able to generalize to unseen scenarios and unseen numbers of objects even if the system has only been trained on a single fixed number of objects.

(c) We show that missing structures in scenarios lead to the collapse of accuracy and what impact this has on task planners.

4.2. GNNs beat NNs in performance and efficiency

Both the used GNN and the used NN use the same data, approximately similar hyperparameters and have a similar network structure with a similar number of possible parameters. However, as can be seen in figure 6, the GNN has a much better performance right from the start and can increase it significantly over several epochs. The NN model can catch up to the initial performance of the GNN, but it cannot improve the test accuracy over many epochs. To increase the accuracy, the network structure would have to be changed significantly and the number of parameters would have to be increased, which leads to more training effort. The NN has a test accuracy of approximately 80% while the GNN can achieve an accuracy of over 95%. The GNN has a much higher scatter than the NN across different seeds, but still performs better with this scatter. As a reference, an input-independent baseline model was trained, which always received zero vectors as input and the respective ground truth labels as target labels. This gives an insight into the accuracy of a random model trained without inputs and shows what the possible achievable accuracy is by pure guessing on the data. The baseline model achieved an accuracy of about 62%. Since the accuracies of both the NN and GNN are much higher than this baseline accuracy it appears they have learned the problem using the training data. The learned models have achieved a significant increase in accuracy, with the GNN showing more efficient training behavior as well as achieving better results. The GNN can make much better use of the structure of the data and better combine individual features through message passing. However, this also leads to more floating point operations in this case. GNNs are therefore superior to classical NN architectures in this considered case not only because of the flexible number of objects.

4.3. The generalization ability of the GNN

The GNN was only trained on nearly 700 graphs with five objects. However, it shows that the system also performs approximately well on scenes whose number of objects the network has not seen. In figure 5 it can be seen how the system achieves an accuracy of over 85% even for the number of objects from 6 to 15. This accuracy was achieved on each 1000 of newly generated data points. This shows how good the generalization performance of the system is. It managed to handle three times the number of objects almost as well. Also, no loss in performance is visible over the increasing number of objects, but it remains at a similar level. Thus, the generalization is also achieved with a strong stability and reliability.

4.4. Lack of structure leads to loss of performance

However, it was also noticed in the tests that the system can handle less than five objects much worse and can only reach the performance of guessing with one object. The missing structure of many objects that have to be left standing as well as moved to intermediate positions results in the system's performance suffering. The GNN has problems with few nodes within the graph, since less exchange between nodes takes place. In the case of the single node, there is no input via the combined feature vectors of the neighboring nodes, so these weights do not provide any input.

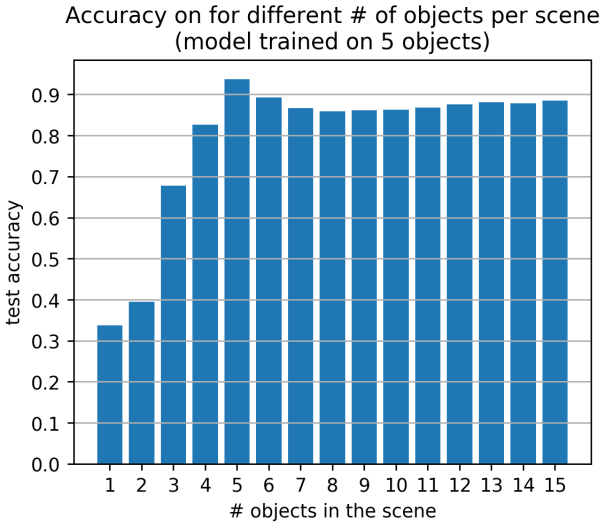

Figure 5. Accuracy of GNN trained on dataset with 5 objects per scene applied to unseen new datasets (different seed) with different numbers of objects per scene.

5. Conclusion and Future Work

In this work, a system for task planning with graph neural networks was implemented. Trained only on 700 scenes with 5 objects each, the GNN has shown absolutely stable zero-shot generalization performance on 1000 scenes with 15 objects. It was shown how GNN outperform normal NN in such tasks and show better and faster training performance. The potential of GNN in the area of task planning was shown.

In future it would be interesting to train different types of GNNs for this problem to see if they perform differently, especially in the generalization to more objects in the scene than trained on.

To improve the generalization of the GNN to a variable number of objects per scene the datasets could be populated by scenes with multiple numbers of objects.

Also a better intermediate position management could be implemented, where the intermediate positions are chosen

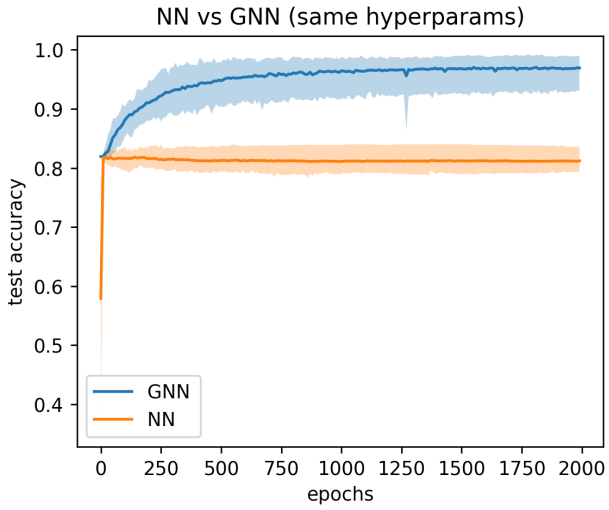

Figure 6. Test set accuracy (min, mean, and max of 10 seeds) of NN and GNN trained on the same data and with the same learning rate of $3e{-}3$.

such that e.g. the travel distance of the robotic arm is minimized. Right now they are placed on the far end of the table in a fixed free space. In an optimized scenario, objects would be placed anywhere they fit without interfering with the other objects and their goal positions.

An obvious future step is to complete the simulation pipeline by connecting the pretrained vision network with the task planning network and the environment and robot simulation to see how the GNN works in a loop.

In the future, the performance of the task planning system needs to be reviewed and tested throughout the pipeline. The simulations in CoppeliaSim should be replaced with links to reality and a real robot arm. Also the Azure Kinect DK camera (Microsoft-Azure) can be used to detect real data and situations.

There is always a difference between simulation and the real world, which could weaken the results of this paper and the usage of GNNs for task planning. For example if there turned out to be catastrophically wrong decisions among the expected errors of the model, it might be necessary to adapt the loss function to reflect the gravity of the single classification decisions and it is unclear how the GNN would perform in such a different scenario.

References

Calli, B., Singh, A., Bruce, J., Walsman, A., Konolige, K., Srinivasa, S., Abbeel, P., and Dollar, A. M. Yale-cmu-berkeley dataset for robotic manipulation research. *The International Journal of Robotics Research*, 36(3):261–268, 2017.

CoppeliaRobotics. Coppeliasim. URL https://www.coppeliarobotics.com/coppeliaSim.

Driess, D., Ha, J.-S., and Toussaint, M. Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image. *arXiv preprint arXiv:2006.05398*, 2020.

Fox, M. and Long, D. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 12 2003. doi: 10.1613/jair.1129.

IAS, I. A. S. Ikida ros. URL https://git.ias.informatik.tu-darmstadt.de/ikida.

Lin, Y., Wang, A. S., and Rai, A. Efficient and interpretable robot manipulation with graph neural networks. *arXiv preprint arXiv:2102.13177*, 2021.

Microsoft-Azure. Azure kinect dk. URL https://azure.microsoft.com/de-de/services/kinect-dk.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4602–4609, 2019.

ORI, O. R. I. Panda arm. URL https://ori.ox.ac.uk/robots/panda-arm/.

Ren, T., Chalvatzaki, G., and Peters, J. Extended task and motion planning of long-horizon robot manipulation. *CoRR*, abs/2103.05456, 2021. URL https://arxiv.org/abs/2103.05456.

ROS, R. O. S. Robot operating system. URL https://www.ros.org/.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Silver, T., Chitnis, R., Curtis, A., Tenenbaum, J., Lozano-Perez, T., and Kaelbling, L. P. Planning with learned object importance in large problem instances using graph neural networks. *arXiv preprint arXiv:2009.05613*, 2020.

Tremblay, J., To, T., Sundaralingam, B., Xiang, Y., Fox, D., and Birchfield, S. Deep object pose estimation for semantic robotic grasping of household objects. In *Conference on Robot Learning (CoRL)*, 2018. URL https://arxiv.org/abs/1809.10790.