# Pick and Place blockstacking task resolution using Graph Neural Networks

Emanuele Coletta

Sapienza University of Rome

Academic Year 2025/2026

# Project objective

Replicate results from the two papers:

- Efficient and Interpretable Robot Manipulation With Graph Neural Networks (Lin et al.)
- One-Shot Imitation Learning With Graph Neural Networks for Pick-and-Place (Di Felice et al.)

Pick and place tasks: assume to have a robot manipulator that can execute a motion primitive which, given an object $o$ in some position $p_o$ and a goal position $p_g$, bring $o$ to $p_g$.

Assume the scene ($K$ object + $L$ goals) is encoded as a graph. Train a GNN policy that chooses which blocks to *pick* and there to *place* them.

Replicate results on block stacking: consider a tower of $K$ objects (blocks) at some position $(x_s, y_s)$. The objective is to invert the tower at some other position $(x_g, y_g)$.

# Scene encoding

A snapshot of the environment is taken after the execution of each *PickAndPlace* task (and at the very start).

A snapshot of the environment is encoded as a fully connected, undirected graph. Each node represents either an object (here object≡block) or a goal. Each node has a feature vector of dimension 6:

- Categorical feature: whether it's a block or a goal (dim=2, one-hot encoded)
- Objective feature: Encodes if a block is inside a goal or if a goal is filled, respectively (dim=1, binary)
- Object position in the robot base frame (dim=3)

# Dataset generation and network training

The dataset is synthetic, obtained by solving randomly generated examples. Each entry of the dataset is the graph form of the scene; the associated labels are the actions to be taken (blocks to pick and the goal to choose) one-hot encoded as for a classification problem. Each example is "perfect": no examples in which a block is misplaced and the policy needs to recover from errors.

Following the references, the network has 3 hidden message-passing layers of 64 nodes each, plus one final output layer that assigns a scalar to each node ( $K + L$ -dimensional output). The nodes are then divided into two $K-$ and $L-$ dimensional outputs, interpreted as the probability of choosing that block/goal. This allows generalization to bigger graphs without changing the structure.
The network is trained to minimize the CE loss on each output.

# Test accuracy

The policy trained on examples with $K = L = 3, 4$ blocks generalizes with 100% accuracy to scenes with up to $K = 6$ blocks, then accuracy starts to lower.
Accuracy gets better with smaller blocks, indicating a data normalization problem.

| $K = L$ | Object acc. | Goal ac.. |
|---|---|---|
| 3 | 1.0000 | 1.0000 |
| 4 | 1.0000 | 1.0000 |
| 5 | 1.0000 | 1.0000 |
| 6 | 1.0000 | 1.0000 |
| 7 | 0.8600 | 0.8600 |
| 8 | 0.8750 | 0.8750 |
| 9 | 0.7822 | 0.7822 |
| 10 | 0.7940 | 0.7940 |

Table: `block_size`=0.2[m]

| $K = L$ | Object acc. | Goal ac.. |
|---|---|---|
| 3 | 1.0000 | 1.0000 |
| 4 | 1.0000 | 1.0000 |
| 5 | 1.0000 | 1.0000 |
| 6 | 1.0000 | 1.0000 |
| 7 | 1.0000 | 1.0000 |
| 8 | 1.0000 | 1.0000 |
| 9 | 0.9089 | 0.9089 |
| 10 | 0.9000 | 0.9000 |

Table: `block_size`=0.05[m]

# Full-scene test accuracy

Ability to solve full scenes should be considered as a metric as well (the policy could choose correctly some blocks/goals, make a mistake, and never solve the rest.)

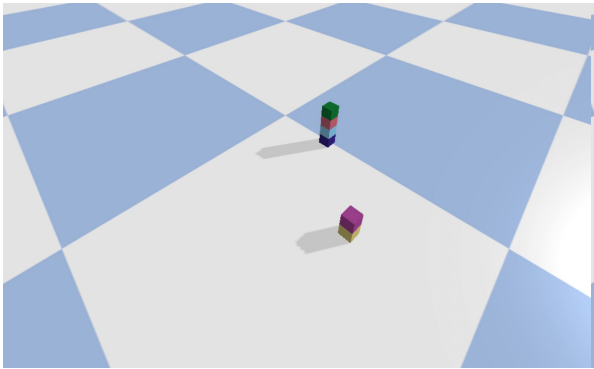


Figure: Test with $K = 6$ (100% test accuracy) - 10/10 scenes solved

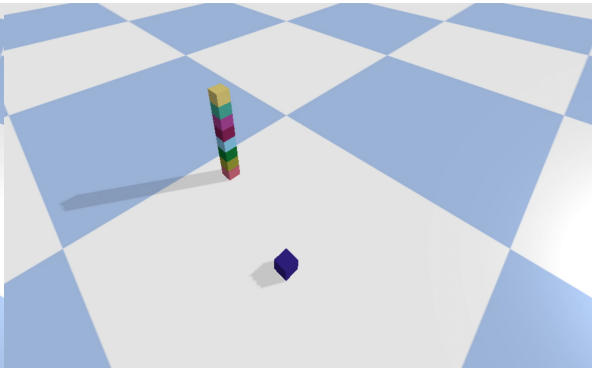


Figure: Test with $K = 9$ (90% test accuracy) - 0/10 scenes solved

# Fine-tuning and problems analysis

With fine tuning with just two additional examples with $K = 8, 10$, the policy generalize up to $K = 12$ blocks.

- The network is robust to small position variations (e.g. perception errors)
- The network fails to generalize to unseen block sizes. This could possibly be resolved either:
  - by normalizing the inputs
  - by creating datasets with more than one block size
  - by using a directed graph that encodes the "on top of" relationship, making the network independent of exact z position