Universität Stuttgart

Institute for Visualization and Interactive Systems

Pfaffenwaldring 5a
70569 Stuttgart

Bachelorarbeit

Inferring Other Agents' Goal in Collaborative Environments Using Graphs

Ruben Werbke

Studiengang: Softwaretechnik

1. Prüfer: Prof. Dr. Andreas Bulling

2. Prüfer:

Betreuer: Dr. Lei Shi, Matteo Bortoletto

begonnen am: 09.10.2023

beendet am: 09.04.2024

Abstract

After the arrival of AI in smartphones, assistance systems and many Internet applications, it slowly makes its way to embodied agents, for example, self-driving cars. While physically manifested agents are uncommon and limited to special use cases, it is important to investigate how agents can be implemented to successfully cooperate in spaces with other participants. To test the social intelligence of agents, the Watch-and-Help challenge was presented, in which AI agents work together with humans in shared apartments. During the watch phase of this challenge, the agent must infer another actor's goal as they perform a common household task. Originally, the agent perceived its environment using a transformer to encode the state of the apartment. In this work, we tested if it was possible to replace the transformer with a Graph Neural Network, which are capable of encoding environments. We further investigate how different encoded relations in the environment graphs affect the capabilities of our new model, discuss its current problems, and propose approaches how to deal with these.

Contents

1 Introduction 13

2 Motivation 15

3 Related Work 17
- 3.1 Collaborative Environments 17
- 3.2 Graph Neural Networks 19

4 Method 21
- 4.1 Overview 21

5 Dataset 25
- 5.1 Watch-And-Help Data 25
- 5.2 Our Data 26

6 Experiments 29
- 6.1 Network Design and Hyper Parameters 29
- 6.2 Over Fitting 32
- 6.3 Impact of Used Edges 32

7 Discussion 35

8 Results 37

Bibliography 39

List of Figures

4.1 Overview of the Goal Inference Module 21

5.1 Environment at time step 0 27
5.2 Final data sample construct 28

6.1 Loss Plot of first training attempts (smoothed) 31
6.2 Class distribution in training set explains the model's tendency towards a single class 31
6.3 Loss of training and validation 32
6.4 Training loss compared to different edges used during training (smoothed, faded plots are unsmoothed data) 33

List of Tables

4.1 Overview of data shape 22

5.1 Features in the Final Data Set 28

6.1 Main Network Layouts 30
6.2 Test results for different used edge types 34
6.3 Testing with all edges on the train set 34

List of Abbreviations

AI Artifical Intelligence. 15, 17

GIM Goal Inference Module. 21, 22, 29, 30, 32, 35, 37

GNN Graph Neural Network. 15, 21, 32, 35, 37

LSTM Long Short Term Memory. 21, 22, 29, 30, 32

MEG Multi Edge Graph Convolutional Network. 21, 22, 30, 32

PyG PyTorch Geometric. 22, 26, 32

WAH Watch-And-Help. 15, 16, 17, 18, 21, 23, 25, 26, 28, 29, 35, 37

1 Introduction

With the fast development of artificial intelligence, we can think of more and more places where to use it. Over time, many “smart”-devices, that are connected to the internet (of things), moved into everybody’s life. While in the beginning the devices were mostly still following strict algorithms (for example a vacuuming robot cleaning a flat), nowadays many devices involve some sort of artificial intelligence (like Google’s Photo App or ChatGPT). Thinking further ahead, robotic assistants might help with physical work in shared spaces alongside other robots or humans. To do so efficiently, these robots can not only work on their own but require some sort of collaboration. This becomes trickier if the agents in an environment are not all of the same type or if the environment involves humans as well. Xavier Puig and his colleagues developed a virtual environment in which it is possible to put different agents together (AI agents as well as human players)[PRB+18]. These can then simulate living together in a shared apartment which also involves a series of different chores that can be carried out. Based on this simulator, Puig and his colleagues then proposed the Watch-and-Help challenge [PSL+20], which consists of two phases. First, the watch phase in which an agent watches another participant working on a task. During this phase, the agent tries to infer what goal is trying to be achieved. Once the agent figured out the goal, it should be able to help achieve this goal in the next phase – the help phase.

During this work, we focused solely on the watch phase and the “Goal Inference Module” which is used in this phase. The Goal Inference Module from the original paper uses a transformer to encode the state of the environment at each time step. One time step denotes a single action like taking up a plate from a table or moving from one room to another room. As environments can easily be described using an environment graph, we thought of replacing the transformer with a graph neural network. This means, all the entities in the flat are described as nodes and their relations as edges. By this, edges could represent a “touching”, “on” or “under” relation as in the predicate: “The glass is on the table”. While the main goal is to replace the transformer, we also take a look at how different edge types affect the performance of the model.

2 Motivation

Artifical Intelligence (AI) is currently showing up in more and more spaces in everyday life. While we need to learn how to work and live with AI, we also have to design AI agents in a way that allows them to work with us - not only for us. Puig and his colleagues came up with the Watch-And-Help (WAH) challenge [PSL+20], which yields a baseline for collaborative environments in which AI agents and humans live and work together. As we will see in the next section, there were more papers published under the topic of collaborative environments for AI agents. All the selected papers share the restriction that the "games" (or challenges) take place in partially observable environments. For conceiving their environment, the different agents were implemented using various techniques.

On the other hand, all the research projects we found preliminary to our own research, were using transformers to encode their environments. Depending on the environment encoding, a transformer is very well suited for that job. However, in many cases, the environment is already encoded as a graph. Graphs are a way to represent data in a broad variety of topics [FGS98]. They allow to encode a lot of information into their topology while still being intuitive. From describing chemical compounds over visualizing social interactions in groups to depicting maps as a data structure, graphs have taken place in many different applications [FGS98]. For many applications, a benefit of graphs is the simple representation of meaningful relations between entities. For example, to render a scene, binary tree search in a scene graph allows one to quickly determine which objects are relevant for the current calculation and which can be ignored as they are out of view [Zac77]. Chemical compounds and molecules are naturally encoded in graphs. Not only is it possible to infer properties of a molecule knowing only the structure with the help of a Graph Neural Network (GNN). Successful experiments have shown that a GNN can build a chemical compound upon a given prompt, which will have all demanded properties.

As GNNs proved themselves as a powerful type of neural network and have successfully been used, we were wondering if it was possible (and beneficial) to replace the transformer that is used in the WAH papers' goal inference module with a GNN. We could encode different information of the environment into a graph such as the type and state of an entity and its relation to its geometrical neighbors. With the use of GNNs we

could work on either node-focused or graph-focused level. In terms of WAH, we could either solely encode the state of the flat, as it was done in the original paper with the transformer, or infer the performed task from the global state of the GNN.

3 Related Work

3.1 Collaborative Environments

A lot of research has gone into collaborative environments for AI agents, where they interact with each other, the environment, or both. We find significant differences in how the challenges are posed and, as a result, widely varying ways how agents are implemented. AI-AI collaboration is a field of ongoing research where many different branches have been explored. Putting AI agents into video games is a common way to train and test them either for working together as a team or to challenge each other as opponents [SDIM19] [LWT+17].

In [SRD+19], Samvelyan and his colleagues use the video game StarCraft as an environment to train and test their agents. StarCraft is a real-time strategy game in which one faction fights another faction. For this, every faction has several units which can be controlled individually. This defines the agent's goal: fight members of the other faction until there are no more. Every agent was given control over one individual unit. In the game world, each agent only has a restricted field of view, limited in range but not obscured by objects in the environment. They did not have any knowledge about foes or team members outside their view range. The policy for the agents yielded rewards for taking out enemies and punished getting hit. This means the degree of collaboration between agents of one team is very limited. Looking at the policy, it could be considered a greedy approach to winning the game without having a global plan that would require any further collaboration between the individual units. However, the perception of the agents is similar to what the agents in WAH were given.

More interaction between different units can be found in [SDIM19]. For this paper, Suarez and his colleagues built a game to investigate large-scale multi-agent interaction. The goal of each agent is, to survive as long as possible. This requires an agent to collect water and food in regular intervals, with food being a limited resource and water being only available in a few spaces. Agents are capable of attacking other agents to fight for food. In different experiments, they either had all agents sharing the same model parameters or had them organized into species. The latter shared model parameters only within the species. Each agent had a limited view range but was able to create its own map of the environment from places it had visited. While this environment does

not make the agents engage in teamwork, the results prove that agents adjust to what other agents do. This manifests in heat maps illustrating what areas are more or less frequently visited, showing that agents tend to explore more and venture further away from denser populated areas than they would do otherwise.

We see the most collaboration between AI agents in [BKM+19], where agents played hide-and-seek in teams of up to three. The goal for the hider team is to stay undetected for the limited time of the game and hence, the seekers must find at least one hider to win the game. Different from the former two publications but similar to WAH, here the agents were also capable of using entities of the world in a more complex way than removing them (as picking up food in [SDIM19]). They were able to move entities or lock them in place, which led to different behavior in both teams, depending on what members of the other team did to the environment. Similar to the WAH agents, agents didn't have global knowledge about the environment during training however, for optimization, an encoded state of the entire world was used without masking. All agents of one team shared the same policy parameters but acted differently due to their individual observations.

On a slightly different note, we find [LPB+19] exploring a way how to learn an activity from watching its execution. The goal is to extract the essence of a task from watching it or having a high-level description of it so that later an agent would be able to execute the tasks again, potentially also in a different environment. The agent therefore needs to first understand the basic steps required to perform the task and then adapt these in a different environment. An environmental graph is used to describe the flat at every time step, where every edge represents a spatial relationship between the objects in the flat. To encode the environment, this graph is fed into a GNN, capturing object relations and allowing it to infer a state for further processing.

Besides [LPB+19], most research focuses on collaboration between AI agents rather than how those agents work aside humans. The absence of research into this direction was also noticed in [MFB+21]. Möller and her colleagues explore how robots in real life, physically in the same space as a human, and not simulated in an virtual environment, have to be designed to ensure a safe and useful coexistence. They state that, to have robots physically exist alongside humans, many different challenges have to be tackled. However, these challenges belong to a broad bandwidth of research topics, which is why they focus on navigation for socially compliant robots. Even though this sounds very specific, the approaches presented in the paper can be of interest to any kind of socially compliant robot, as they involve understanding human behavior and acting upon it.

3.2 Graph Neural Networks

First introduced in [GMS05], GNNs were a new way to work directly on a graph without further preprocessing. Getting rid of the need to encode a graph into a flat vector first, the risk of loosing information embedded into the topology of the graph was eliminated. By now, GNNs are represented in very different applications, reaching from computer graphics, over natural language processing and chemistry to traffic control and prediction [WPC+20].

Extending the work of [MFB+21], Ravichandran and his colleagues did research on how to use GNNs to encode three dimensional space in a way that explains it to an agent [RPH+22]. Using a high-level representation of the environment in the form of a scene graph, their model is capable of learning navigation policies to move around the scene and keep track of its trajectory. They prove that their model manages to search and find different objects in its environment

Improving the performance of agents during search tasks in environments like households, [KLA+23] came up with a solution that is capable of predicting the location of different objects. This is achieved by using a combination of two different techniques, first using a Scene Graph Memory for exploring the environment and secondly a Node Edge Predictor to predict new node edge combinations. Utilizing the Scene Graph Memory, the agent collects several observations of the environment over a certain period. Once familiar with the environment, it can then be tasked to find an object in the environment (that was not seen before or might have moved since it has been seen) and find it using the Node Edge Predictor.

4 Method

4.1 Overview

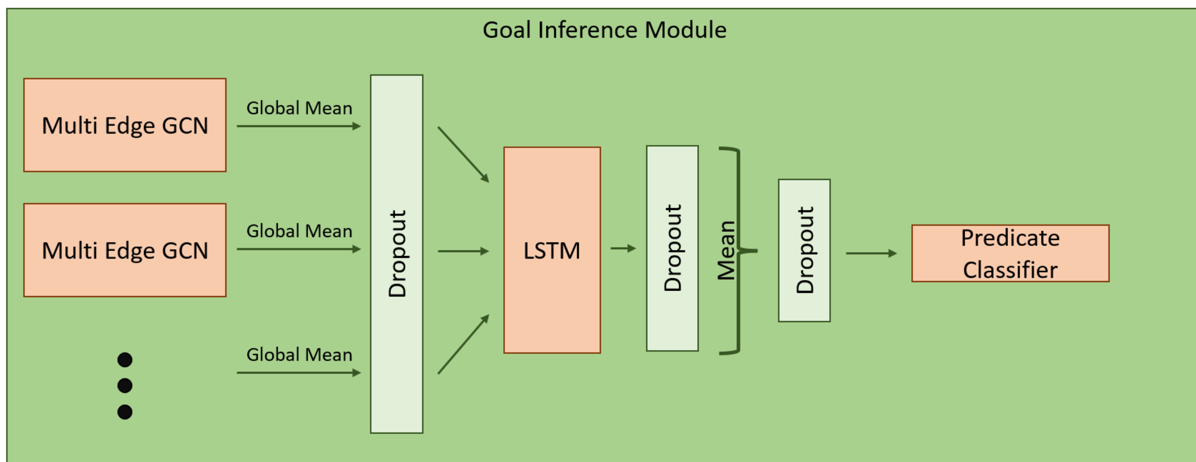


Figure 4.1: Overview of the Goal Inference Module

To replace the transformer with a GNN, the data needed a different encoding than used in the WAH paper. This different style required us to update some parts of the Goal Inference Module (GIM) to work with different dimensions and keys in the set.

Our implementation consists of four main components: three different modules (orange in 4.1) intertwined inside the GIM (dark green in 4.1). The first part is the Multi Edge Graph Convolutional Network (MEG), which is the replacement of the transformer as it was used in the WAH implementation. After this comes a Long Short Term Memory (LSTM), collecting the outputs from the MEG per time step. The GIM is reshaping the data around these two first modules and implementing more basic layers to improve the output of our network. The last component is the Predicate Classifier, which, in its initial form, was copied from the WAH implementation.

Snapshot	Data Shape
Before MEG	List of Graphs (see 5.2)
After MEG	tensor(batch_size · num_nodes, features)
Before LSTM	tensor(batch_size, features)
After Predicate Classifier	tensor(batch_size, num_classes)

Table 4.1: Overview of data shape

4.1.1 Goal Inference Module

The input to the GIM is a "demonstration", as it is described in 5.2. Before the data can be used in the MEG, it has to be reshaped. PyTorch adds a batching dimension as the first dimension of the data, which does not work in PyG. PyG implements batching by concatenating the adjacency matrices of each sample in the batch in a diagonal manner. This style of batching can be achieved by unpacking the batched data from PyTorch and then adding it each individually into a PyG "Batch" wrapper. After transformation, we can feed the graphs (for each time step) into the MEG, computing the final state of each node in each graph. The data returned from the MEG is a tensor that still has the batch dimension as PyG uses it. Before proceeding, this gets reshaped to match PyTorch's style of batching (compare "After MEG" and "Before LSTM" in 4.1). The global state of each graph is then calculated using global mean pooling (as described in [WPC+20]). Finally, using the Predicate Classifier, the likeliness for each class is inferred. Three dropout layers were added to the GIM, one after the global mean pooling, one after the LSTM, and another one after the mean pooling. The dropout layers were supposed to improve the overfitting problem this network has and have been kept when the experiments got frozen even though the overfitting could not be significantly improved as discussed in the "Results" section.

4.1.2 Multi Edge Graph Convolutional Network

This module receives a node index and an edge index as input. While we are only using one type of node at this point, which has its type encoded in its feature vectors, the edges vary in type (see section 5.1 for more details on this). During its forward call, the MEG channels the data through three graph convolutional layers, each followed by a ReLu activation and finally through a fully connected linear layer. The most important part about this module is, that, right after initialization, it gets transformed to a heterogeneous graph capable version using PyGs transform function. This performs

graph convolutions on each sub-graph (edge type wise) and shares those results with other sub-graphs each iteration.

4.1.3 Predicate Classifier

Originally copied from the WAH code, this module was responsible for counting how often every predicate showed up during a demonstration. As we were not able to run the original code on our systems (due to memory requirements) and we could not infer all intentions in the code from looking at it, we were not able to understand the functionality of this last module to the point where we could reproduce it entirely. This is why we ended up using the predicate classifier to predict the performed task rather than counting the predicate occurrences during the demonstration. Its implementation however has not been modified any further than changing its output dimension (shown in 4.1).

5 Dataset

5.1 Watch-And-Help Data

For their experiments, Puig and his colleagues used data that was created using the Virtual Home Simulator [PRB+18]. The scenarios this simulator offers were created using descriptions of common tasks from everyday life, collected from a variety of people. From this, a sample was selected to be used for training and validation of the watch phase in the WAH paper.

This sample offers seven different apartments of different sizes and layouts. Every flat consists of at least one of each of these rooms: Bathroom, Kitchen, Bedroom, and Living Room. Between different apartments, rooms of the same kind differ in their fit-out, size and connection to adjacent rooms. To populate the rooms, 136 different assets are available, split up into 25 categories (for example Lamps, Furniture, Decor, and Appliances). Every asset can be used several times per flat, to make a flat look like a real-life flat. For training and validation of the goal inference during the watch-phase, demonstrations are used. A demonstration shows the execution of a specific task to the point where it is considered completed. Five different tasks are available, namely `read_book`, `put_dishwasher`, `prepare_food`, `put_fridge` and `setup_table`. Each task is part of a demonstration in several of the different apartments or, for those demonstrations taking place in the same flat, has objects placed in different parts of the environment. It is also possible that the goal of the demonstration, even though it is the same task, might look different. For example, `setup_table` requires forks and knives on the table for one demonstration, but in a different demonstration it only needs knives on the table.

The data was split into a training and a validation set consisting of 5303 demonstrations for training and 1021 demonstrations for the validation set. Out of the seven apartments mentioned before, only five were used in the training demonstrations. Two were held back for the validation set, which is not only using those extra two flats but all seven environments. This is to ensure, that the model is capable of inferring a task not only in known environments but also in an unknown apartment.

5.2 Our Data

Using the code from the WAH project, we generated the dataset, making sure the split between training and validation data is the same. While originally the two datasets were two objects that were kept in memory during the entire runtime, we could not do the same due to the object's size and the memory limits on our machines. Thus, all demonstrations were extracted from the original sets and saved, each individually, as binary files for later access.

The goal then was, to generate a dataset that is compatible with PyTorchs data loaders, lightweight on memory, accessible and modifiable, and working well together with the modules from PyTorch and PyTorch Geometric (PyG) with as little as possible modifications on the fly. Each demonstration, as we received and saved it from the WAH code, contained a series of information of which we took the `task_name` and the `graphs`. While there was more information encoded in the data, due to little documentation, we were not able to infer its use up to a point where it could have become useful for us. Opposed to the `task_name`, which is a human-readable descriptor of the depicted demonstration (matching one of the five task names named before), the `graphs` field does not hold any graphs. Instead, it is a list of flat states, in which each element represents one time-step throughout the demonstration. The flat states are encoded as a list of objects, in which each object describes an entity in a flat or an entire room (without its content). Every entity has a category, a name, valid states and properties assigned as attributes as well as a bounding box object, which holds the position and extent of the entity.

5.2.1 Edges and Nodes

We use the entities' attributes to generate nodes and edges. Every demonstration holds one ordered list per attribute: `class_names`, `categories`, `properties` and `states`, which each holds all possible values for all nodes over the entire dataset (not only the demonstration).

For every node, the feature vector is generated from a series of node specific attribute vectors. Every attribute vector has as many dimensions as the corresponding list. The entries in the specific attribute vector are 1 if the corresponding node holds the attribute at the given index and 0 otherwise. All four feature vectors for the specific node get concatenated to become the nodes feature vector. In section 5.2.2, we explain why all features over the entire dataset are needed during this construction and not only the ones from the current demonstration.

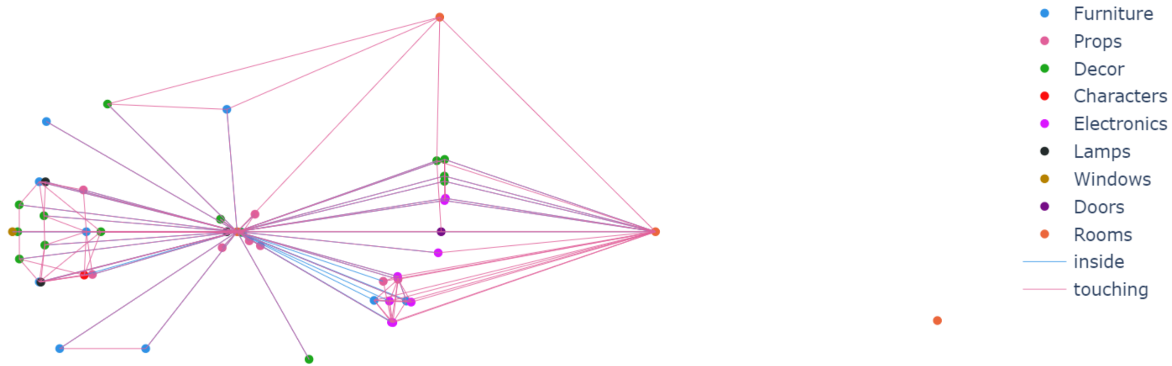


Figure 5.1: Environment at time step 0

The bounding boxes were solely used to create edges. Five different edge types were computed during the creation of the dataset, representing "inside", "touching", "is below", "is above" and "is close" relations between nodes. To refine the "inside" relation, additional restrictions based on the classes of the two nodes in question were made, reducing the edge count by 16%. The other four relations are purely based on the position and extent of the nodes. For the "is close" relation, a threshold of 1.5 meters was chosen, which is approximately the length from one hand to another at fully stretched arms. This relation was added, as it could help to decide whether an object is at its final position already or not. It also helps with the "inside" relation, which in some cases connected a node to two rooms, as their bounding boxes overlapped too much (depicted in 5.1).

5.2.2 Padding

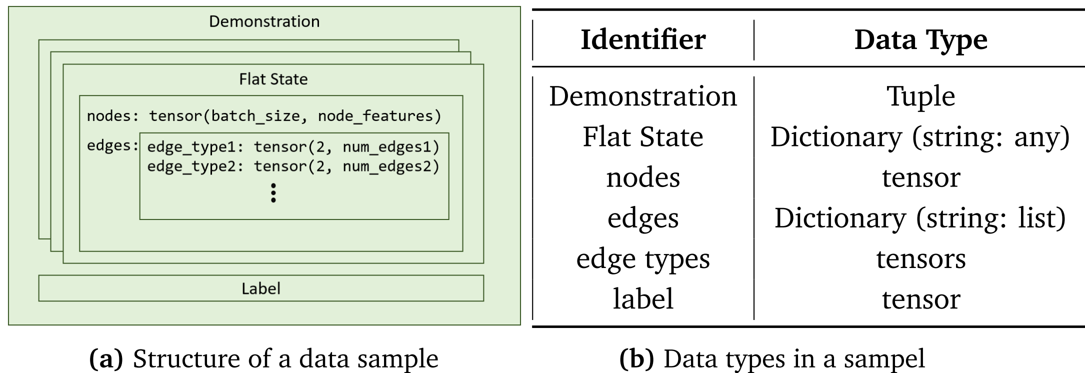
PyTorch needs every sample or batch of samples to be of the same size for its modules to work correctly. As the demonstrations vary in length between 50 and 80 time steps, the number of graphs per demonstration would vary between this range too. Further, depending on which apartment is used in the demonstration, the number of rooms and objects in those rooms varies too and thus, the number of edges per graph varies significantly too. As this would yield very differently sized dimensions between different demonstrations, all graphs needed to be padded. For shorter demonstrations, padding graphs were needed to fill up the otherwise empty time steps at the end. The feature vector of a padding node contains -1 at every position and padding edges all went from node 0 to node 0.

5.2.3 Final Data Set

Constructing the data set as described, using all available data from WAH, yields demonstrations with graphs as described in 5.1. Loading one sample from the dataset returns a tuple as described in 5.2. The label is encoded as a one-hot tensor. The index of the 1 indicates the corresponding class among "read book", "put dishwasher", "prepare food", "put fridge" and "setup table".

Number of Graphs	80
Number of Nodes	135
Number of Node Features	187
Number of Edges	inside: 138 touching: 340 below: 77 above: 176 close: 1128

Table 5.1: Features in the Final Data Set



Identifier	Data Type
Demonstration	Tuple
Flat State	Dictionary (string: any)
nodes	tensor
edges	Dictionary (string: list)
edge types	tensors
label	tensor

(a) Structure of a data sample **(b)** Data types in a sampel

Figure 5.2: Final data sample construct

6 Experiments

Several different experiments have been conducted during the process of developing and testing the new GIM. The entire process can be split into three sections. During each section, a series of tests was conducted to evaluate the current step. In 6.1, we give a coarse overview of how the network design developed over time. This does not reflect every single change, which will be discussed in the different subsections, but rather shows the main snapshots during development. Network 3 corresponds to the network presented in 4.1 and was only added to the table to allow comparison to the previous implementations.

6.1 Network Design and Hyper Parameters

During this early stage of the development of the new GIM, we focused on achieving an uninterrupted dataflow. Therfore we evaluated the training loss and adjusted the layout of the network and the hyperparameters. Network 1 (see 6.1) was the first network that allowed the data to flow through without any dimension issues and provided an uninterrupted gradient flow during optimization. However, there was no decrease in the loss plot.

We realized that a hidden size of 64 was way too small to capture all details in the data (compare 5.1: number of graphs · number of nodes · number of node features). Another issues with Network 1 was the interpretation of how to use the output from the LSTM. PyTorchs LSTM returns a tensor in shape (batch_size, time_steps, features). In Network 1, only the last time step had been used (due to a misunderstanding in how LSTMs work), while in fact, the mean over all time steps (as it was also done in the code from WAH) was needed.

After fixing the mentioned issues, several training runs were started to investigate whether the network was learning but without any success. The loss fluctuated with up to $\pm 50\%$ but did on average not improve at all, as shown in 6.1. Besides checking that the optimizer worked, several attempts were made to improve the results, testing different combinations of learning rates, annealing and activation functions of different types (ReLu and Tanh) behind different layers. Plotting the prediction of the network

finally gave an insight into what the issue with the training was. Looking at the model's predictions (6.2b) showed that it always leaned towards classes 2 and 3 (noticeable over several different training runs). Plotting the occurrences of the tasks from the training set suggested that the model can not deal with the highly unbalanced training set. After reading [SGL+24], we tested training with different weights for each class during the loss calculation. However, we found that this technique was not sufficient for our purposes. Another mentioned approach suggested to balance the set by copying samples from underrepresented classes or removing samples from overrepresented classes. To quickly achieve results, we went with the latter-mentioned solution, where we picked the first 600 samples of each class. With these updates, the loss plot looked like the model was learning.

Network	**Design**	
	Module	Parameters
Network 1	GIM	no dropouts using only last layer of LSTM
	MEG	hidden/output layers: 64 internal pooling and aggregation of sub-graphs
	LSTM	hidden/output layers: 64 hidden layers: 2
Network 2	GIM	no dropouts mean over all LSTM layers linear layer after LSTM
	MEG	hidden/output layers: 512 internal pooling and aggregation of sub-graphs
	LSTM	hidden/output layers: 512 hidden layers: 3
Network 3	GIM	added dropouts inferring global state from MEGs output
	MEG	hidden/output layers: 512 no internal pooling
	LSTM	hidden/output layers: 512 hidden layers: 2

Table 6.1: Main Network Layouts

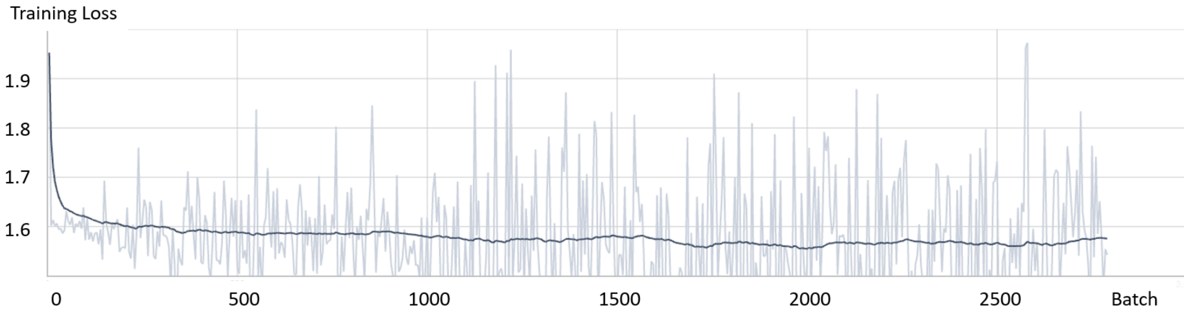
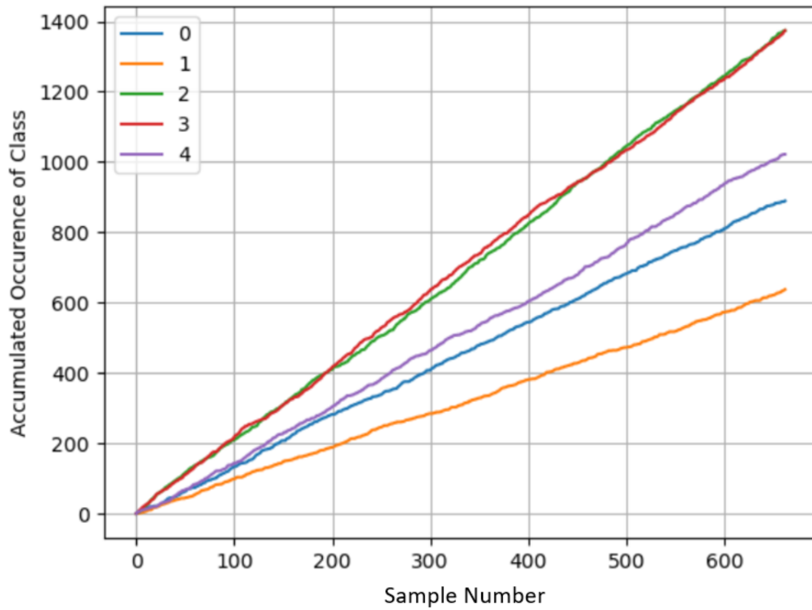
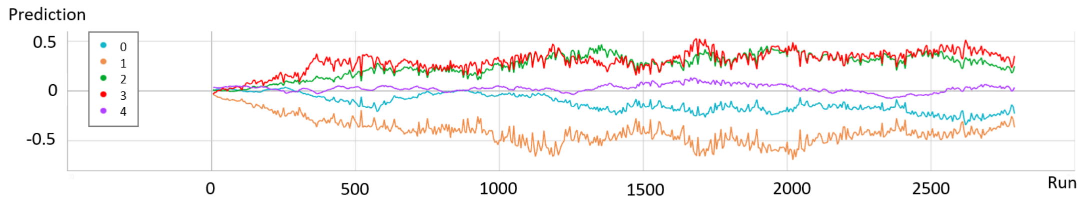


Figure 6.1: Loss Plot of first training attempts (smoothed)



(a) Distribution of classes in training set



(b) Predicted likeliness of each class per sample

Figure 6.2: Class distribution in training set explains the model's tendency towards a single class

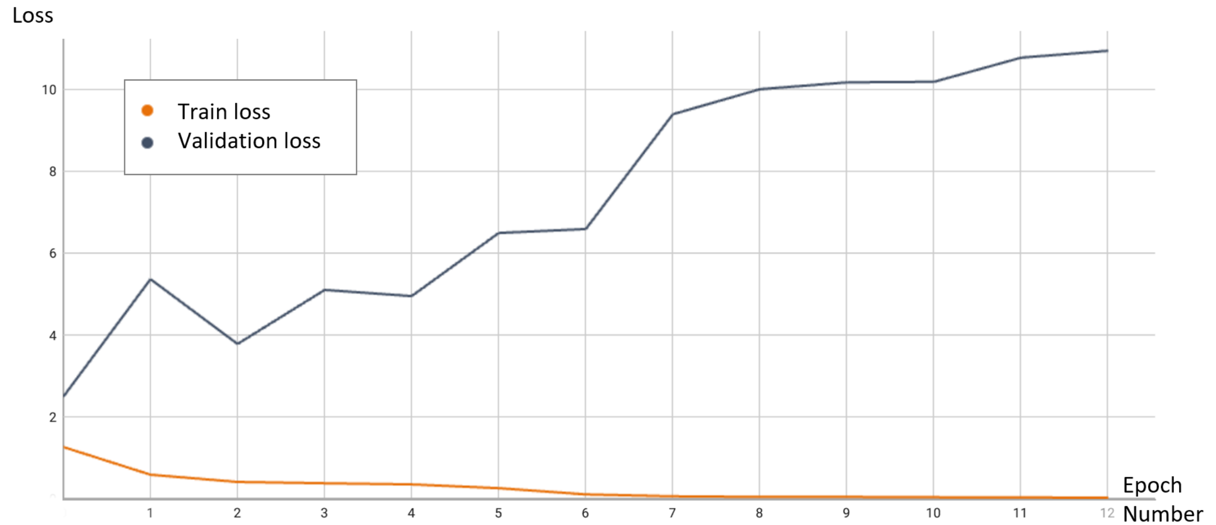


Figure 6.3: Loss of training and validation

6.2 Over Fitting

Using Network 2 (from 6.1), the plotted loss decreased during training, but the validation loss did not (see 6.3). As a first measure, using new insights in PyG, the MEG was redesigned making use of PyGs internal transformation from GNNs for homogeneous graphs to one for heterogeneous graphs. Another linear layer was added to the GIM with the intention that the new depth of the network would allow it to generalize more and reduce the overfitting by that.

After 20 rounds of training, the most promising results were achieved with Network 3 (see 6.1, which is also the one described in 4.1). Even though during these 20 rounds of training, a wide range of different settings were tested, both for the hyperparameters as well as some internal parameters of the network (dropout probabilities, hidden LSTM layer numbers, adding and removing of linear layers between other layers), no improvement for the validation could be found.

6.3 Impact of Used Edges

As a last experiment and disregarding the overfitting issue the current model has, we looked at whether using different edges for training makes a difference. Therefore, we ran four different rounds of training with the same network and all equal hyperparameters. The edges we used were:

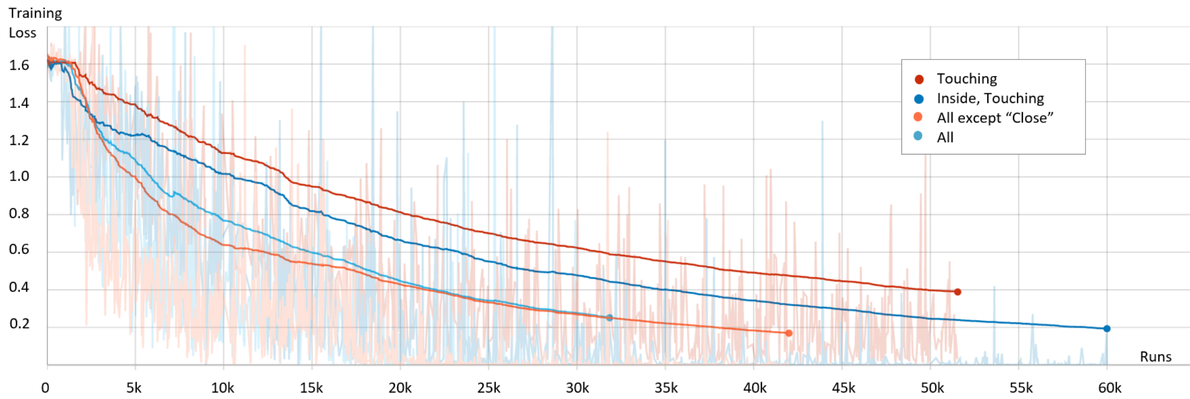


Figure 6.4: Training loss compared to different edges used during training (smoothed, faded plots are unsmoothed data)

1. all edges
2. inside and touching edges
3. only touching edge
4. all edges besides the "is close" relation

We plotted the loss functions during training as it can be seen in 6.4. This already suggests that the types of used edges matters. For a deeper discussion of the findings, refer to section 7.

Edges	Precision	
	Per Class	Average
touching	0, 0.28, 0.12, 0, 0.42	0.16
inside, touching	0, 0.23, 0.0, 0, 0	0.05
all except "is close"	0, 0.38, 0.12, 0.0, 0.32	0.16
all	0.89, 0, 0.15, 0, 0.44	0.30

(a)

Edges	Recall	
	Per Class	Average
touching	0.0, 0.66, 0.0043, 0.0, 0.10	0.15
inside, touching	0.0, 0.82, 0.0, 0.0, 0.0	0.16
all except "is close"	0.0, 0.66, 0.0043, 0.0, 0.10	0.15
all	0.08, 0.0, 0.05, 0.0, 0.72	0.17

(b)

Edges	Correct Prediction
touching	30%
inside, touching	22%
all except "is close"	30%
all	42%

(c)

Table 6.2: Test results for different used edge types

Precision per class	0.96, 0.0, 0.96, 0.83, 0.56
Precision average	0.67
Recall per class	0.44, 0.0, 0.47, 0.65, 0.73
Recall average	0.56

Table 6.3: Testing with all edges on the train set

7 Discussion

In [PSL+20], the GIM achieves precision and recall of 0.85 and 0.96 during testing. Compared to this, our model can not keep up with the original GIM. The best results our model achieved were precision and recall of 0.30 and 0.17 (averaged over the different classes) using all five edge types. The results from the WAH paper however look at the "predicate counts" and do not infer the overall task type as we did (we described the reason for this in section 4.1.3). As mentioned in the introduction, a predicate is something like "One fork on the table", hence guessing the task type can be considered a more general approach to inferring the agent goal rather than counting the predicates. For a closer comparison between a GNN and the transformer, the overfitting problem needs to be overcome first.

Different techniques can be used to reduce overfitting. Due to the time spent coming up with the current layout of the model and understanding the problem during training caused by the unbalanced training set, we did not have time to test these approaches. One way could be the implementation of more layers to allow the model to generalize more, in combination with refitted dropout layers or residual blocks. Another possibility would be to change how we deal with the unbalanced data set. Instead of removing samples from over-represented classes as it is done at the moment, copies from demonstrations of underrepresented classes could be inserted. Doing so would increase the overall amount of training data and thus help the model to generalize more.

Changing the way the graphs and demonstrations were padded could also yield different results. Instead of adding the padding edges from and to the first node, these could be added to the last node, which would likely be a padding node as well.

It would be interesting to investigate the impact of different edge types in the environmental graphs. While we focused on very basic relations during the development of our model, the used edge types could have a big impact on the model's performance (as the results listed in 6.4 suggest). Besides using spatial relationships, logical relations like "can use" could be beneficial.

8 Results

We built a new GIM using a GNN to replace the transformer. In its current implementation however, we have to admit, that our GIM can not compete with the network presented in WAH. As 6.3 shows, the model is capable of learning the different classes in the balanced training set and is not simply guessing labels. This suggests that, with more development, the model should be able to achieve better precision and recall than it has so far. In 7, we mentioned possible solutions to solve the issues (most importantly the overfitting) of the current GIM. Comparing precision and recall depending on which edge types were used, shows, that the selection of used edge relations impacts the models' performance.

Bibliography

[BKM+19] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, I. Mordatch. “Emergent tool use from multi-agent autocurricula.” In: *arXiv preprint arXiv:1909.07528* (2019) (cit. on p. 18).

[FGS98] P. Frasconi, M. Gori, A. Sperduti. “A general framework for adaptive processing of data structures.” In: *IEEE transactions on Neural Networks* 9.5 (1998), pp. 768–786 (cit. on p. 15).

[GMS05] M. Gori, G. Monfardini, F. Scarselli. “A new model for learning in graph domains.” In: *Proceedings. 2005 IEEE international joint conference on neural networks, 2005.* Vol. 2. IEEE. 2005, pp. 729–734 (cit. on p. 19).

[KLA+23] A. Kurenkov, M. Lingelbach, T. Agarwal, E. Jin, C. Li, R. Zhang, L. Fei-Fei, J. Wu, S. Savarese, R. Martın-Martın. “Modeling dynamic environments with scene graph memory.” In: *International Conference on Machine Learning*. PMLR. 2023, pp. 17976–17993 (cit. on p. 19).

[LPB+19] Y.-H. Liao, X. Puig, M. Boben, A. Torralba, S. Fidler. “Synthesizing environment-aware activities via activity sketches.” In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 6291–6299 (cit. on p. 18).

[LWT+17] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, I. Mordatch. “Multi-agent actor-critic for mixed cooperative-competitive environments.” In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 17).

[MFB+21] R. Möller, A. Furnari, S. Battiato, A. Härmä, G. M. Farinella. “A survey on human-aware robot navigation.” In: *Robotics and Autonomous Systems* 145 (2021), p. 103837 (cit. on pp. 18, 19).

[PRB+18] X. Puig, K. Ra, M. Boben, J. Li, T. Wang, S. Fidler, A. Torralba. “Virtualhome: Simulating household activities via programs.” In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8494–8502 (cit. on pp. 13, 25).

[PSL+20] X. Puig, T. Shu, S. Li, Z. Wang, Y.-H. Liao, J. B. Tenenbaum, S. Fidler, A. Torralba. “Watch-and-help: A challenge for social perception and human-ai collaboration.” In: *arXiv preprint arXiv:2010.09890* (2020) (cit. on pp. 13, 15, 35).

[RPH+22] Z. Ravichandran, L. Peng, N. Hughes, J. D. Griffith, L. Carlone. “Hierarchical representations and explicit memory: Learning effective navigation policies on 3d scene graphs using graph neural networks.” In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 9272–9279 (cit. on p. 19).

[SDIM19] J. Suarez, Y. Du, P. Isola, I. Mordatch. “Neural MMO: A massively multiagent game environment for training and evaluating intelligent agents.” In: *arXiv preprint arXiv:1903.00784* (2019) (cit. on pp. 17, 18).

[SGL+24] R. Shwartz-Ziv, M. Goldblum, Y. Li, C. B. Bruss, A. G. Wilson. “Simplifying Neural Network Training Under Class Imbalance.” In: *Advances in Neural Information Processing Systems* 36 (2024) (cit. on p. 30).

[SRD+19] M. Samvelyan, T. Rashid, C. S. De Witt, G. Farquhar, N. Nardelli, T. G. Rudner, C.-M. Hung, P. H. Torr, J. Foerster, S. Whiteson. “The starcraft multi-agent challenge.” In: *arXiv preprint arXiv:1902.04043* (2019) (cit. on p. 17).

[WPC+20] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, S. Y. Philip. “A comprehensive survey on graph neural networks.” In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24 (cit. on pp. 19, 22).

[Zac77] W. W. Zachary. “An information flow model for conflict and fission in small groups.” In: *Journal of anthropological research* 33.4 (1977), pp. 452–473 (cit. on p. 15).

All links were last followed on 3rd April 2024.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature