CHALMERS
UNIVERSITY OF TECHNOLOGY

Learning-Based Task Assignment for Automated Guided Vehicles

Applying graph neural networks to optimise task assignment in an online warehouse environment

Master's thesis in Electrical Engineering

Axel Borgvall
Nils Ivarsson

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Learning-Based Task Assignment for Automated Guided Vehicles

Applying graph neural networks to optimise task assignment in an online warehouse environment

Axel Borgvall
Nils Ivarsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of systems and control
Automation research group
Chalmers University of Technology
Gothenburg, Sweden 2026

Learning-Based Task Assignment for Automated Guided Vehicles
Applying graph neural networks to optimise task assignment in an online warehouse environment
Axel Borgvall, Nils Ivarsson

Supervisor: Sabino Francesco Roselli, Department of Electrical Engineering
Examiner: Knut Åkesson, Department of Electrical Engineering
Industrial Supervisor: Fredrik Johansson, MAXAGV

Master's Thesis 2026
Department of Electrical Engineering
Division of systems and control
Automation research group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: MAXAGV model FX32R automated guided vehicle.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Learning-Based Task Assignment for Automated Guided Vehicles
Applying graph neural networks to optimise task assignment in an online warehouse environment
Axel Borgvall, Nils Ivarsson
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Efficient task assignment in multi-Automated Guided Vehicle (AGV) warehouse environments is critical for optimizing industrial logistics. In collaboration with MAX-AGV, this thesis evaluates the application of reinforcement learning (RL) to address this challenge. The warehouse environment is modelled as a graph, and a Graph Neural Network (GNN) policy is trained using Proximal Policy Optimization (PPO) to assign tasks to the vehicle fleet. To capture the complex topology of the facility, which is characterized by long-range spatial configurations and lock-relations that limit standard embedding methods like Node2Vec, a novel transductive node embedding scheme trained via multiple task-specific decoders is introduced. Three core GNN architectures: Graph Convolutional Networks (GCN), Graph Attention Networks (GAT), and Graph Transformers, along with their heterogeneous extensions, are evaluated and compared against conventional heuristic baselines. The empirical results demonstrate the performance trade-offs between the learning-based architectures and traditional heuristics. Furthermore, the study addresses the broader challenges of deployment, specifically the complexities of reward shaping in real-world logistics systems and the systemic barriers to integrating learning-based methods into legacy industrial infrastructures.

Keywords: AGV, Neural Networks, GNN, Attention, Simulation, Graph Embeddings, Reinforcement Learning, PPO, Task Assignment, Warehouse Automation

Acknowledgements

We would like to thank our supervisor Sabino Francesco Roselli for his invaluable support throughout the course of this project. His guidance in shaping the direction of the work, consistent feedback, and regular meetings were instrumental in keeping the project on track. We are also grateful for access to the hardware he provided, which made the training and evaluation of our models possible. We also thank our examiner Knut Åkesson for his role in overseeing the work. A particular thanks goes to Alvin Combrink for his valuable input on the graph neural network design and for helping us develop our understanding of how to approach the problem. Finally, we would like to thank Fredrik Johansson and MAXAGV for providing the collaboration opportunity and for their support throughout the project.

Axel Borgvall, Nils Ivarsson, Gothenburg, June 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AGV	Automated Guided Vehicle
EMA	Exponential Moving Average
GAE	Generalised Advantage Estimation
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
GT	Graph Transformer
MAPF	Multi-Agent Path Finding
MDP	Markov Decision Process
MLP	Multi-Layer Perceptron
MPNN	Message Passing Neural Network
NN	Neural Network
PPO	Proximal Policy Optimisation
RL	Reinforcement Learning
TAN	Task Assignment Network

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Graph notation

G	Graph
$\mathcal{V}$	Set of nodes
$\mathcal{E}$	Set of edges
$\mathcal{N}(i)$	Neighbourhood of node i
x_i	Feature vector of node i
e_{ij}	Edge attribute between nodes i and j
$\deg(i)$	Degree of node i

GNN notation

t	Layer index
m_i^{t+1}	Aggregated message at node i in layer $t+1$
W	Learnable weight matrix
b	Learnable bias vector
a	Learnable attention vector
P_d	Projected node embedding dimension
G_d	GNN output embedding dimension

Reinforcement learning

s_t	State at time t
a_t	Action at time t

r_t	Reward at time t
π	Policy
θ	Policy parameters
$V(s)$	Value function of state s
$\hat{V}(s)$	Estimated value function
γ	Discount factor
λ	GAE parameter
δ_t	TD residual at time t
$\hat{A}_t$	Advantage estimate at time t
R_t	Return target at time t
$r_t(\theta)$	Probability ratio between new and old policy
ϵ	PPO clipping parameter
c_1	Value loss coefficient
c_2	Entropy coefficient
T	Rollout length
E	Epochs per iteration
B	Mini-batch size

Embeddings

$\tilde{L}$	Symmetric normalised Laplacian
$\hat{A}$	Symmetrically normalised adjacency matrix
D	Degree matrix
k_{conn}	Connectivity embedding dimension
k_{lock}	Lock relation embedding dimension

Contents

List of Figures

List of Tables

1

Introduction

The use of automated guided vehicles (AGVs) has been steadily rising over the last few years [1], and is projected to continue growing over the coming decade [2]. AGVs are increasingly used in automated warehouses, improving productivity and operational flexibility [1]. The management of an AGV fleet is a non-trivial optimisation problem, which can be divided into two subproblems: task assignment and routing. Task assignment is the process of deciding which AGV should execute which task, while routing is the process of planning collision-free paths for each AGV to its assigned task. These subproblems can either be addressed separately, or jointly as a multi-agent pathfinding (MAPF) problem [3], to which numerous solutions have been proposed [4, 5]. In practice, the two subproblems are often treated independently, with routing handled by a dedicated system. This separation allows each subproblem to be addressed with dedicated methods, enabling more precise optimisation of each component independently. Accordingly, this thesis focuses on the decoupled formulation of the problem, seeking the best possible solution to the task assignment problem assuming an existing routing algorithm.

1.1 MAXAGV

The work presented in this thesis is carried out in collaboration with MAXAGV, a company specialising in the design, development and manufacturing of AGVs and accompanying software solutions [6]. The company traces its roots back to the Volvo Kalmar plant in the 1970s, where the automotive industry first started adopting the use of AGVs. Today MAXAGV serves a broad global customer network across numerous industry sectors, with active systems deployed in more than 25 countries. MAXAGV designs and develops all products in-house, including a wide array of AGV robots, systems for converting existing forklifts into fully autonomous versions, and the MAX software which serves as the primary fleet management platform. The MAX software is the system responsible for routing and task scheduling in the AGV fleet, as well as integrating with on-site warehouse hardware such as conveyors and warehouse management systems.

The MAX software handles the pairing of individual tasks to individual AGVs and the planning of each of their paths separately. Consequently, integrating a system that solves for both task assignment and path planning concurrently with the MAX system would be architecturally infeasible. This thesis therefore focuses on devel-

oping a task assignment solution that maximises performance given MAXAGV's existing pathfinding system.

1.2 Scope and delimitations

The thesis focuses on the theoretical framework and feasibility of using reinforcement learning to train a policy for AGV task assignment. The approach is evaluated in a simulated warehouse environment, where the learned policy is assessed against conventional heuristic baselines. The following research questions are investigated:

1. To what extent can a reinforcement learning-based policy, represented by a graph neural network, match or surpass conventional heuristic methods in AGV task assignment?
2. How does the choice of graph neural network architecture influence the training stability and task assignment performance of the learned policy?
3. What reward function is required for effective AGV task assignment, and how does the reward function's design affect the generalisability of the learned policy?
4. What methods can be used to create descriptive agent inputs which accurately encode the complex topology and features of real-world AGV facilities?

The aim of this thesis is not to produce a deployment-ready task assignment system. Practical concerns such as hardware integration and AGV fleet management are outside the scope of this work. Alternative assignment methods are not investigated in depth. Conventional heuristics are instead used solely as comparative baselines against which the performance of the learned policy is evaluated.

1.3 Related works

Several studies have investigated reinforcement learning for task assignment in multi-robot systems, including the GRAND network [7] and the RTAW network [8]. Both are designed to output high-level robot-task assignments rather than fully planned paths and execution strategies for the assigned pairs. Specifically, the GRAND network utilises a GNN where nodes represent robots and tasks, and the edges correspond to possible paths between them. The spatial information of the robot nodes is encoded within the input node vectors. Similarly, the MAGNNET implementation [9] employs an RL-trained GNN. This approach models the input as a bipartite graph with tasks and robots as nodes, utilising proximal policy optimisation as the optimisation algorithm.

A notable distinction between prior studies and this thesis is the general topology of the environment. Prior studies frequently use grid-like environments or similarly path dense environment topologies. Such environments provide greater flexibility in path selection between two points and stresses traffic resolution in the algorithm

used. Real-world storage facilities, however, are often structured around one-way lanes branching off into many corridors, see Figure 2.2a, resulting in significantly sparser layouts than the topologies considered in prior literature.

1.4 Thesis outline

Chapter 2 presents the theoretical background relevant to the methodology and discussion. Chapter 3 describes the simulation environment used for training the reinforcement learning agent. Chapter 4 details the methods used to generate node embeddings that serve as input to the agent. Chapter 5 outlines the network architecture, training pipeline, and evaluation strategy. Chapter 6 presents the results together with preliminary insights, followed by a thorough discussion in Chapter 7.

2

Theory

This chapter presents the theoretical background necessary to understand the methods used in this thesis. It covers node embeddings, graph neural networks, and reinforcement learning, which together provide the framework for training the agent.

2.1 Task-agent scheduling

In the context of a multi-AGV system, task-agent scheduling is the process of determining which agent should do what at any given point. In this case the term is taken to mean both deciding which agent should be given which specific task as well as deciding the exact paths the AGVs will travel to their destinations.

2.1.1 Task assignment

Task assignment refers to the allocation of tasks to individual AGVs. For instance, given a fleet of AGVs A and B alongside a set of tasks 1 through 5, AGV A may be assigned to task 1 while AGV B may be assigned to task 4. This process is limited to task allocation. Subsequent decisions, such as path planning and routing, are excluded. As illustrated in Figure 2.1, while task assignment designates the destination node 11 for AGV A (originating at node 0), the vehicle may still navigate via multiple distinct paths, such as the green or red paths.

In this thesis, task assignment consists of pairing up each AGV with a task selected from a queue of uncompleted tasks. Each task is defined by a two-phase sequence: navigating to a designated pickup location to retrieve a payload, followed by transit to a secondary delivery location.

The task assignment problem is in reality often quite complicated. Naive heuristics, such as assigning each AGV to its nearest task, often have unintended side-effects which may lead to reduced system performance. For instance, such strategies can result in AGV crowding and blocking each other, thereby increasing idle times as AGVs wait for routing conflicts to resolve. Additionally, assignments may become suboptimal when tasks are reassigned to closer AGVs after initial allocation, resulting in wasted time and energy spent travelling in vain.

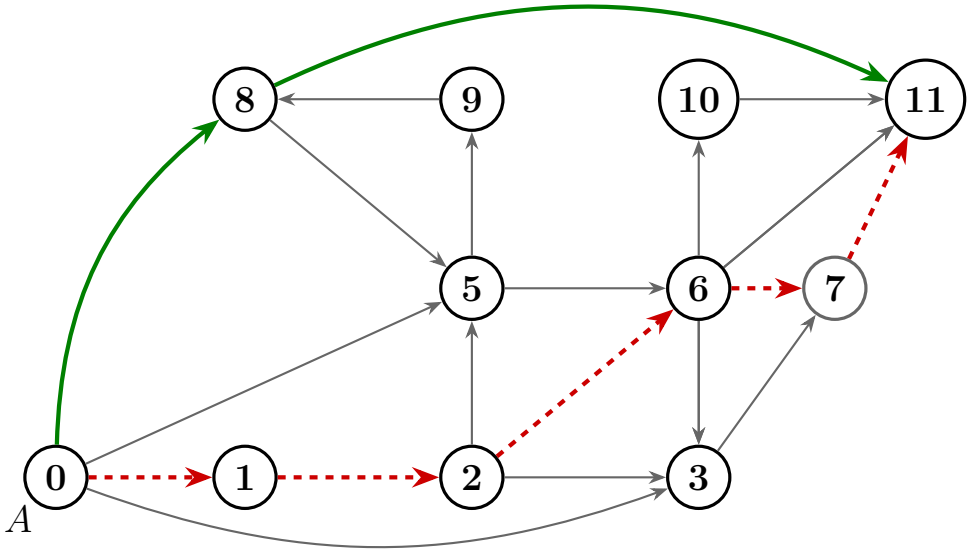
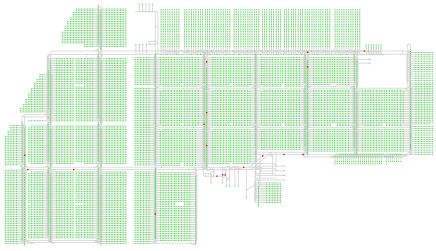


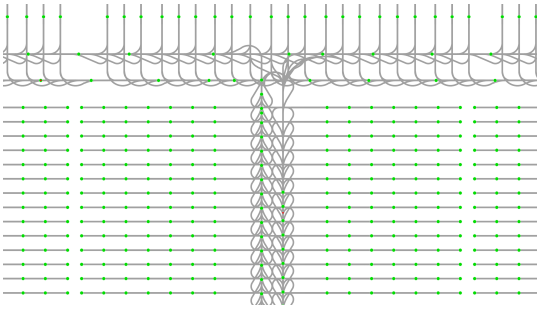
Figure 2.1: Example of different paths between the same source and destination nodes in a graph. One path goes along the top of the graph and the other along the bottom.

2.2 Graph representation of a storage facility

In the AGV systems examined in this thesis, AGVs move along predetermined paths divided into unidirectional segments. The connections between segments are predefined, i.e. whether or not an AGV is allowed to switch from segment i to segment j upon reaching the end of i. This means the storage facility can be represented as a graph wherein the segments are nodes and the allowed switches can be represented as edges between them.



(a) Image of the graph layout of a facility using AGVs. Green dots are stations, red dots AGVs and grey lines are the segments the AGVs are permitted to travel along.



(b) Zoomed in view of the facility layout. The segments follow much more intricate and fine patterns than what is visible from the view of the full layout.

Another aspect of the facilities handled in this thesis is so called lock relations. In order to prevent AGVs from taking paths that would cause them to collide or get in the way of one another, a large number of lock relations are defined for the segments in the facility. Each lock relation consists of a lock type, a locking segment and a locked segment. The type defines when the lock is active

- onExit: active as long as an AGV is anywhere on the locking segment.
- onLeave: active if an AGV is on the locking segment and has not yet reached the end of it.
- onDistance: active if an AGV is on the locking segment and has not yet traversed a specific distance along it. The distance is specified individually for each of the onDistance lock relations.

If a segment is locked AGVs are not permitted to enter it and must instead wait or step onto a different segment. The number of lock relations per segment varies wildly, the average number of segments locked by a locking segment is about 200 and some segments lock over a 1000 other segments when an AGV is on them.

2.2.1 Stations

AGVs perform tasks, such as picking up and dropping of loads, at stations. Each station has a number of entering segments from which an AGV can enter the station and perform a task there. The segments entering a station are not necessarily close to each other in graph space. For example just because segment i and j both enter the same station doesn't mean there is a short path from one to the other that an AGV is permitted to travel.

2.3 Node embeddings

In order to build an effective neural network for task assignment it is important to have an input format that well encodes the information the network might need to make its decision. Since our environment consist of a graph on which the AGVs and the tasks to which they can be assigned occupy different nodes it is fitting to create an embedding for each node that encodes its relation to the rest of the graph. The network can then be fed the embedding vectors associated with each of the agent and task positions in the input, giving it representative information about the state of the graph in a format suitable for a neural network.

2.3.1 Node2vec

Node2Vec is an algorithmic framework that encodes the topological structure of a graph into vector representation, enabling a compact and informative representation of node relationships [10]. The method first performs biased random walks from each node to collect information about its neighbours. A dataset is formed from the sequences of nodes from these walks which is then used to trained a skip-gram

model[11], whose learned embeddings form the final vector representation of the nodes.

2.3.2 Decoder trained embeddings

Transductive embeddings is a term for embeddings that are computed prior to training and which do not change. Since the layout of any facility the AGVs operate in is static the embeddings being unchanging is not a downside for this use case. The result is effectively a lookup table of vectors for all the nodes in the graph. In this case our method consisted of a randomly initialized set of embeddings for all segments as well as all stations along with 6 decoders set to compute specific metrics that were deemed relevant to the input. 4 of the decoders are dedicated to training just the segment embeddings and 2 to training both the segment and station embeddings.

- Distance decoder: a multi-layer perceptron with 2 linear layers. Given an input of 2 concatenated embedding vectors it's tasked with computing the time from the first to the second node represented by the vectors. The loss is taken as the mean squared error of the true and predicted time.
- Degree decoder: single layer perceptron. Given an input vector it is tasked with predicting the in/out degrees of edges representing lock relation and connections seperately.
- Lock relation decoder: Given two input vectors it performs elementwise multiplication and applies an SLP to the resulting vector to determine wether or not the two inputs share a lock relation. The loss taken as binary cross entropy.
- On path decoder: an MLP with 2 linear layers. Given an input of 3 vectors concatenated its tasked with computing wether the middle vector lies on the shortest path between the first and third vectors. Loss is taken as binary cross entropy.
- On path to station decoder: Same as the on path decoder except the destination is a station instead of a segment.
- Distance to station decoder: Same as the distance decoder except the distance is taken between a segment and station instead of two segments.

All of these metrics, except for segment degree, are highly compressive. This means the embeddings are forced to find patterns in the data to store the information as effectively as possible rather than just memorizing it.

2.3.3 Laplacian + community embeddings

As an alternative to the trained approach, a closed-form embedding can be derived directly from the two graphs. Due to the differences between the two graphs a different method was used for computing the embeddings of each graph. The two embeddings were then combined with concatenation. Both methods used allow for the length of the embeddings vector to be selected meaning that the data in the

embedding dedicated to each of the two graphs can be freely selected.

The first component is a Laplacian positional encoding computed from the connectivity graph. The symmetric normalised Laplacian $\tilde{L} = I - D^{-1/2}AD^{-1/2}$ is formed from the undirected version of the connection graph, and its k_{conn} eigenvectors corresponding to the smallest non-trivial eigenvalues are retained. Here k_{conn} refers to the length of the embedding vectors formed from the connectivity edges. The k eigenvectors corresponding to the smallest non trivial eigenvalues are the solutions to the constrained optimization

$$\min_{f_k \perp \mathcal{S}_k \ \& \ ||f||=1} \sum_{(i,j)\in\mathcal{E}} \left(f_k(i) - f_k(j)\right)^2 \tag{2.1}$$

Where $f_k(i)$ is the ith element of the kth eigenvector, $\mathcal{E}$ is the set of all edges, $\mathcal{S}_k$ is the set of all previous eigenvectors as well as $\mathbf{1}$. Orthogonality to $\mathbf{1}$ constraints it to ignore the 0th eigenvector with all constant elements and $||f|| = 1$ prevents the trivial solution of $\mathbf{0}$. These elements are then optimized to be as similar between neighboring nodes as possible and to be orthogonal.

To compute the community embeddings for the lock relation edges the symmetrically normalized community matrix is first computed: $\hat{A} = D^{-1/2}AD^{-1/2}$, $\hat{A}_{ij} = \frac{1}{\sqrt{d_i d_j}}, (i,j)||(j,i) \in \epsilon$ otherwise 0. A truncated SVD of rank k_{lock} is then applied to the symmetrically normalised adjacency matrix of the undirected lock graph, retaining $U\Sigma$ as the node representation. This factorisation approximates the leading community structure of the lock graph: nodes that frequently share lock dependencies are embedded close together, while structurally distant nodes are separated.

The two components are concatenated to form a single d-dimensional vector per node, where $d = k_{\text{conn}} + k_{\text{lock}}$. Because both components are computed deterministically from the graph topology, the method is fully reproducible and incurs no training cost beyond a single eigendecomposition and SVD.

2.4 Graph neural networks

A graph neural network (GNN) is a class of neural networks designed to operate directly on graph-structured data [12]. A graph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of nodes and $\mathcal{E}$ the set of edges encoding relationships between nodes. Each node $i \in \mathcal{V}$ is associated with an initial feature vector $\mathbf{x}_i \in \mathbb{R}^d$, and the neighbourhood of node i, denoted $\mathcal{N}(i)$, is the set of nodes connected to node i by an edge. Edges may additionally carry feature vectors $\mathbf{e}_{ij}$ representing relational attributes between connected nodes.

Most modern GNNs are based on the message-passing neural network (MPNN) framework [13]. An MPNN operates by iteratively updating the node representations by aggregating information from its neighbouring nodes. The node representations are updated in two steps, first the message passing step and then the update step.

The message step consists of two parts: message generation and aggregation. For each neighbour $j \in \mathcal{N}(i)$ a message is computed from the function γ^t that relies on the sender's representation, receiver's representation, and their relationship. The incoming messages are then aggregated using a permutation-invariant function (e.g. sum or mean) to produce the aggregated message $\mathbf{m}_i^{t+1}$. In equation 2.2 the complete equation for creating the aggregate message is shown. t represents the layer the representations belong to.

$$\mathbf{m}_i^{t+1} = \sum_{j \in \mathcal{N}(i)} \gamma^t(\mathbf{x}_i^t, \mathbf{x}_j^t, \mathbf{e}_{ij}) \tag{2.2}$$

In the update step shown in equation 2.3 the aggregated message is combined with the node's feature vector. The function ϕ^t can be any differentiable function that combines $\mathbf{x}_i^t$ with $\mathbf{m}_i^{t+1}$ to create the new node representation $\mathbf{x}_i^{t+1}$.

$$\mathbf{x}_i^{t+1} = \phi^t(\mathbf{x}_i^t, \mathbf{m}_i^{t+1}) \tag{2.3}$$

In the above equations $\mathbf{x}_i^t$ denotes the feature vector of node i at layer t, $\mathbf{e}_{ij}$ is the edge attribute connecting node i with node j.

The following GNNs all follow the MPNN framework described above. The difference lies primarily in the choice of the message and update function, γ and ϕ, particularly in how the contributions of neighbouring nodes are weighted.

2.4.1 Graph convolutional networks

Graph convolutional networks (GCN) combine the framework of MPNN with regular convolutional networks. First described in 2017, it was one of the first MPNNs introduced and used at large scale [14]. GCN's message function resembles that of a convolution. A weight matrix is first applied to the sender's feature vector, followed by a normalisation term consisting of the square roots of the sender's and receiver's node degrees, before a *sum* aggregation function is applied, this can be seen mathematically in equation 2.4 below.

$$\mathbf{m}_i^{t+1} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{\mathbf{W}^t \mathbf{x}_j^t}{\sqrt{\deg(i)}\sqrt{\deg(j)}} \tag{2.4}$$

Where $\mathbf{W}^t$ is the learnable weight matrix and $\deg(i) = |\mathcal{N}(i)| + 1$, to account for the added self-loops. Also, note that when compared with equation 2.2, GCN does not incorporate the edge attributes, limiting it in edge-weighted graphs. GCN also has no possibility to differentiate the weight given to incoming messages, the normalization is only there to scale everything to similar proportions, but can not give different weights to different messages.

For the update step, the GCN update function is simply the identity, replacing the node representation directly with the aggregated message.

$$\mathbf{x}_i^{t+1} = \mathbf{m}_i^{t+1} \tag{2.5}$$

Unlike equation 2.3, the previous feature vector $\mathbf{x}_i^t$ is not explicitly included in the update function, as it is already included through the self-loop in the message passing step.

2.4.2 Graph attention networks

A graph attention network (GAT) is an MPNN introduced in 2018 to address the limitations of GCN [15]. It extends the GCN framework by replacing the fixed normalisation, with learned attention weights. This allows the GAT to assign different levels of importance to different neighbours, making it more expressive than a GCN.

The message function of a GAT, shown in full in equation 2.6 below, computes a weighted sum of the neighbourhood, where the weights are learned attention coefficients.

$$\mathbf{m}_i^{t+1} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \text{softmax}_j \left(\text{LeakyReLU} \left((\mathbf{a}^t)^T \left[\mathbf{W}^t \mathbf{x}_i^t \parallel \mathbf{W}^t \mathbf{x}_j^t \right] \right) \right) \mathbf{W}^t \mathbf{x}_j^t \tag{2.6}$$

In the message function $\mathbf{a}^T[\mathbf{W}\mathbf{x}_i \| \mathbf{W}\mathbf{x}_j]$ computes a raw attention score between node i and a neighbour j, here $\|$ denotes the concat operation. $\mathbf{a}$ is a learnable attention vector and $\mathbf{W}$ is a shared weight matrix that linearly transforms node features into a common space. The LeakyReLU is applied to introduce nonlinearity and the softmax normalises the scores across all neighbours so they sum to one, making them interpretable as importance weights. The final message is then the weighted sum of the transformed neighbour features $\mathbf{W}\mathbf{x}_j$.

$$\mathbf{x}_i^{t+1} = \sigma(\mathbf{m}_i^{t+1}) \tag{2.7}$$

In the update step, equation 2.7, a nonlinearity σ is applied to the aggregated message, such as ELU or ReLU, to increase the expressive power of the network.

2.4.3 Graph transformers

Attention-based neural networks have paved the way for many advances within machine learning, most notably in large language models. The Graph Transformer (GT) architecture extends this paradigm by applying transformer-style attention to graph-structured data. The GT is based on the key, query, and value attention mechanism introduced by Vaswani et al. [16]. A standard transformer takes as input a sequence of some kind, for example a sentence or a time series. In the graph domain, this sequence is instead formed by the neighbours of a given node. Equation 2.8 shows how the attention mechanism is applied in the message step of an MPNN.

$$\mathbf{m}_i^{t+1} = \sum_{j \in \mathcal{N}(i)} \text{softmax}_j \left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}} \right) \mathbf{v}_j \tag{2.8}$$

For each node i a query vector $\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i$ is calculated, representing what the node is looking for. Each neighbour computes a key vector $\mathbf{k}_j = \mathbf{W}_K \mathbf{x}_j$ and a value vector $\mathbf{v}_j = \mathbf{W}_V \mathbf{x}_j$ representing what the neighbour offers and what information it carries respectively. The attention score is then the dot product of $\mathbf{q}_i$ and $\mathbf{k_j}$, scaled with the square root of the key dimension. Softmax is applied to ensure that the attention score is weighted with all scores in mind. Lastly, each neighbour's value vector $\mathbf{v}_j$ is included to send the information. The update function follows that of the original transformer from *Attention is all you need* [16].

$$\mathbf{h}_i = \text{LayerNorm}(\mathbf{x}_i^t + \mathbf{m}_i^{t+1}) \tag{2.9}$$

$$\mathbf{x}_i^{t+1} = \text{LayerNorm}\left(\mathbf{h}_i + \text{FFN}(\mathbf{h}_i)\right) \tag{2.10}$$

The update applies a LayerNorm to the sum of the skip-connection and message. The normalized sum is then passed through a learnable feed forward network (FFN), and again normalized after addition with a skip-connection around the FFN, to compute the new feature vector.

2.5 Reinforcement learning

Reinforcement learning (RL) is a machine learning paradigm focused on sequential decision-making under uncertainty. Rather than learning from a pre-labelled dataset, an agent interacts with an environment, selecting actions to maximise the cumulative reward signal [17]. In contrast to supervised learning, which assumes independent, labelled input-output pairs, RL is characterised by an explicit agent-environment interaction loop: the agent observes the current state, produces an action, and receives a scalar reward, after which the environment transitions to a new state.

The agent is the decision maker that interacts with the environment by selecting actions. The environment represents everything the agent operates in, moving to a new state when the agents selects an action and outputting a reward signal. This interaction creates a closed loop, the agent selects an action a_t, the environment then transition from state s_t to state s_{t+1}, giving a reward of r_t. In state s_{t+1} the agent now selects a new action and the loop continues [17]. The agent's goal is to learn a behaviour that maximises the reward over time.

The way the agent's behaviour changes is based on the *policy* [17]. The policy is a way for the agent to decide what action to take given the current state, a map between states and actions. The policy may be a simple lookup table, a heuristic, or a more advanced neural network.

As the agent's goal is to maximise the accumulated reward, the *reward signal* is critical, it defines what is good and bad behaviour [17]. The reward signal sent to the agent is based on the current environmental state and the agent's action. The only way the agent can influence the reward signal is through its actions. It can either explicitly change the reward by choosing another action in a given state, or implicitly by changing the state of the environment. The reward signal is the main part of the learning, if a selected action gives a low reward, the policy is incentivised to choose another action in the same state.

While the reward function gives an immediate signal of what is good and what is bad. The *value function* signals what is good in the long run [17]. The value of a state s is the accumulated reward an agent can earn from starting in s until the end of the episode. A state may have a low reward but a high value, or vice versa, impacting how desirable states are.

Most reinforcement learning algorithms are grounded in the formalism of a Markov decision process (MDP), which assumes the environment satisfies the Markov property: the next state depends only on the current state and action, not on prior history. An MDP is defined as a four-tuple (S, A, P_a, R_a) [17].

- S is the set of possible states in the environment.
- A is the set of actions, the action space.
- $P_a(s, s')$ is the probability of taking action a in state s and moving to state s'.
- $R_a(s, s')$ is the reward given when taking action a in state s and moving to state s', the reward signal.

In practice, many real-world environments only partially satisfy this assumption, as the agent may not have access to the full state of the environment

2.5.1 Actor-Critic

Actor-critic refers to a family of architectures for training an agent in an RL environment. They are based on the more general policy gradient methods. Policy gradient methods seek to maximise the performance of a specific policy [17]. It updates the policy vector θ by doing gradient ascent on:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.11}$$

where $\nabla J(\theta_t)$ is an approximation of the gradient of the performance measure J to its argument θ_t.

Policy gradient methods focus on learning a policy function. Extending this to also learn an approximation of the value function yields the actor-critic method. The actor refers to the learned policy, as it chooses the actions taken, and the critic refers to the value function, as it judges the actions taken by the actor.

During each interaction with the environment, the actor selects an action a_t, the environment then transitions from state s_t to the new state s_{t+1}, and a reward r_t is observed. The critic then computes a TD error, equation 2.12, which serves as a learning signal for both actor and critic [17].

$$\delta = r_t + \gamma \hat{v}(s_{t+1}, \mathbf{w}) - \hat{v}(s_t, \mathbf{w}) \tag{2.12}$$

$$\mathbf{w} = \mathbf{w} + \alpha_{\mathbf{w}} \delta \nabla \hat{v}(s_t, \mathbf{w}) \tag{2.13}$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \delta \nabla \ln \pi(a_t | s_t, \boldsymbol{\theta}) \tag{2.14}$$

The update rules above describe the one-step actor-critic, where the TD error,δ, uses a single reward and bootstraps from the next state. In practice, multi-step returns or more sophisticated advantage estimates are used to better balance bias and variance. With the guidance of the TD error δ both the value function's parameters $\mathbf{w}$ and the policy's parameters $\boldsymbol{\theta}$ are updated according to equations 2.13 and 2.14. This combination inherits the low variance of TD learning compared to full Monte Carlo returns, at the cost of some bias introduced by bootstrapping from an approximate value function.

2.5.2 Proximal Policy Optimisation

Proximal Policy Optimisation (PPO) is an on-policy actor-critic algorithm that builds on standard policy gradient methods [18]. Standard policy gradient performs a single gradient step for each batch of experience and then discards it, maintaining on-policy correctness at the cost of sample efficiency. PPO addresses this by reusing each batch for multiple gradient steps, while clipping policy updates to a trust region to prevent catastrophically large changes to the policy [18].

PPO has demonstrated strong empirical performance across a wide range of challenging reinforcement learning problems [18]. It takes the idea of trust region methods, which constrain how much the policy can change in a single update, but achieves this through a simpler clipped surrogate objective rather than a hard constraint. PPO has three distinct phases: rollout, advantage calculation, and update.

During the rollout phase, PPO gathers data about the policy and environment. The environment is stepped through for T timestamps using the old policy $\pi_{\theta_{old}}$, collecting the actions taken, value estimates, and rewards received.

PPO requires an accurate advantage estimate across an entire rollout of collected experience. A single-step TD error provides a low variance estimate, but is biased since it relies on the accuracy of the critic. The full Monte Carlo return is unbiased, but suffers from high variance due to the accumulation of stochastic rewards. Generalised Advantage Estimation (GAE) interpolates between these two extremes via an exponentially weighted sum of TD residuals, controlled by $\lambda \in [0, 1]$ [19].

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{2.15}$$

Here δ_{t+l} is the TD residual defined in equation 2.12, and γ is the discount factor that down-weights contributions from distant time steps. The product $\gamma\lambda$ jointly controls how quickly the weights of future TD residuals decay.

During update, PPO steps through the same observed states as during rollout collecting a new policy $\pi_\theta(a_t|s_t)$ and value estimate $\hat{V}_\theta(s_t)$. It then calculates a probability ratio, equation 2.16, between the new and old policy, for the specific action a_t in state s_t. Comparing the updated policy to the policy that collected the data.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.16}$$

Rather than performing a single gradient step on the full rollout, PPO iterates over the collected data for E epochs, sampling minibatches of size B at each step. The probability ratio $r_t(\theta)$ is then computed for each minibatch, comparing the updated policy to the policy that collected the data. The clipped surrogate objective, equation 2.17, restricts how far this ratio may deviate from the old policy, preventing large updates in the wrong direction.

$$L_t^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2.17}$$

The clipping hyperparameter ϵ defines a trust region, if the probability ratio would exceed $[1-\epsilon, 1+\epsilon]$ the objective would be flat in that direction and the gradient zero. A key property of taking the minimum is that it forms a pessimistic lower bound on the unclipped objective, ensuring the policy is not rewarded for moving outside the trust region.

For the value function PPO uses a simple mean squared error between the critic's estimated value $\hat{V}_\theta(s_t)$ of the state and the return target $R_t = \hat{A}_t + \hat{V}_{\theta_{old}}(s_t)$, computed from the GAE advantage estimate.

$$L_t^{VF} = \left(\hat{V}_\theta(s_t) - R_t \right)^2 \tag{2.18}$$

The full PPO objective function combines the policy surrogate with the value function loss and an entropy bonus.

$$L_t(\theta) = \mathbb{E}^t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF} + c_2 S[\pi_\theta](s_t) \right] \tag{2.19}$$

The entropy bonus, $S[\pi_\theta](s_t)$, encourages exploration by penalising premature convergence to near-deterministic policies. The coefficients c_1 and c_2 scale the importance of the critic loss and entropy bonus in the full objective function.

2.6 Theory Summary

In summary, this chapter has introduced the theoretical framework underlying the proposed system. PPO, Section 2.5.2, provides the training algorithm updating the

policy and value network weights. The networks consists of a GNN, Section 2.4, which is well suited for the environment since the relationships between AGVs and tasks are naturally represented as a graph. Three GNN variants are compared in this work, GCN, Section 2.4.1, GAT, Section 2.4.1, and GT, Section 2.4.1. Since individual node features alone may not capture the topological graph structure, two embedding methods are introduced, Sections 2.3.2 and 2.3.3.

3

Simulation Environment

This chapter outlines the high level logic behind the simulation used to train the RL agent. It also includes a section on an alternate environment which was created but which has not yet been used for training or evaluating any actors.

3.1 Simplified simulated environment

An environment was created based on the data relating to a storage facility from layout files provided by MAXAGV. Then the shortest paths and the time taken to traverse the shortest path was precomputed for every segment-station pair. The path from a segment to a station was taken as the path from the segment to the entering segment to that station which could be reached in the shortest possible time. At the start of a simulation run the following is done

1. Every AGV is given a position in terms of segment index and a `time_left` variable describing the time remaining that the AGV has to spend travelling along the segment it is on before it reaches the segment end and can step of it and onto one of the segments it flows into.

2. A buffer of randomly generated tasks is created. Each task is given a status, untouched/picked up/completed, a priority 0-100, a time indicating when it was first created, and two station indices, one for the station where it can be picked up and another for the station where it should be dropped off. The index pairs are drawn from a log of one months worth of real-world transports as to be as realistic as possible.

For every step of the simulation the AGVs move and perform tasks until there is a change in the task buffer which necessitates a new action from the agent. More precisely, every step the simulation

1. Checks whether any AGVs currently loaded are at that tasks designated dropoff station. If so, the AGV leaves the payload at the station and the task's status is updated to completed in the buffer.

2. Checks whether any AGVs currently not carrying a load are at the pickup station associated with their assigned task. If so, the AGV picks the payload up and the task's status is updated to picked up

3. For every AGV the station it should go to is chosen. The pickup or drop-off station depending on whether the AGV has already picked the load associated with its task up or not.

4. All AGVs move along their respective segments for dt seconds, if they are at the end of their segment they stand still.

5. Out of all AGVs that are at the end of their segment they try to step onto the next segment in the shortest path to their destination station and do so if it is not locked. If the next segment in the AGVs path is locked it waits at the end of it's current segment. If an AGV has been stuck for a considerable amount of time it is allowed to look for a longer path to its destination and switch paths if it has an non-locked alternative.

6. If all AGVs have been stuck with no viable segments to move to for a considerable amount of time the environment is marked as deadlocked.

7. The tasks that are completed are removed from the task buffer and it is refilled with new randomly generated tasks.

8. This is repeated until an AGV picks up a task, drops one off and becomes available for reassignment, a task is added to the queue, or a certain number of iterations is reached. The step is then concluded and the agent queried for task reassignment.

There are mechanics of the real simulation which are not accounted for in this version. In the real system some segments are blocked from access unless an AGV is assigned to a task related to a specific station associated with that segment. The real system has a function where an AGV can ignore the segment connections and, under special conditions, take a path not normally allowed in order to resolve a deadlock. There may be many additional rules present in the real system that are not captured by the simulation.

In order to procedurally generate tasks for the actor to choose from, a log of a month's worth of transports for the facility was used to extract pickup and dropoff station pairs from real transport data. The tasks were then procedurally generated during training by sampling from this set of station pairs in order to ensure realistic tasks.

During training multiple simulated environments were run in parallel using the Python multiprocessing package. This sidesteps the GIL that normally makes true parallelism impossible in Python by launching multiple interpreters.

3.2 Wrapping of MAXAGV system into a RL environment

A scaled down version of MAX, the system in use by MAXAGV, was also interfaced with in order to create a RL environment. This was done by creating a *Gymnasium* [20] environment which treated the simulation as a black box. Using a tcp socket to send information back and forth between the backend simulation and the *Gymnasium* environment written in Python.

For every step of the environment the agents action was sent as a *json* of pairings between AGV IDs and task IDs. This version of the simulation also uses so called idle assignments. If an assignment of an AGV to a task is failed, then it is instead assigned to go to the closest idle station. An idle station is in effect just a position the AGV can occupy without being in the way of other AGVs.

4

Segment and station feature embeddings

This chapter describes how the segment and station embeddings used to create the observations fed to the RL agent were created and evaluated. It also includes the method for creating and evaluating an alternate set of embeddings which were used as a comparison for the main decoder trained embeddings.

4.1 Node embedding training

This section outlines the two distinct methodologies used to generate representative embeddings for both layout segments and stations. First, the primary approach using a trained decoder model is described. Second, a baseline approach using static Laplacian and community embeddings is detailed to serve as a comparative benchmark.

4.1.1 Decoder embeddings

The model structure outlined in section 2.3.2 was trained on the same facility the simulation was to be run on. For the on path and distance metrics the same precomputed path lookup table was used as in the simulation. In order to simultaneously train the embeddings to encode all six metrics the six loss items were weighted using Kendall uncertainty weighted loss [21]. This is a method which uses learned parameters to automatically fine tune the weighting of individual loss items when training for multiple metrics. All segment and station embeddings were initialized using Xavier uniform initialization. They were then trained together with the decoders with the losses generated from the decoders.

4.1.2 Laplacian + community embeddings

As the laplacian + community embeddings require no training they were simply computed using the connections and lock relation edges read from the layout used for simulation. The station embeddings were taken as the average of the embeddings of the entering segments. 48 elements were given to the connectivity laplacian portion of the embeddings and the remaining 16 to the lock relation community embeddings.

4.2 Node embedding evaluation

A comparison between the performance of different node embedding schemes when used in actual training is beyond the time scope of this study. Instead the embeddings were graded on how well a number of chosen metrics could be reconstructed from them using small simple learned decoders of the same architecture as was used in training.

4.2.1 Decoder embeddings

In order to make a fair comparison with the laplacian + community embeddings the decoders used during evaluation were trained from scratch on the frozen embeddings rather than using the same decoders as were used during the training of the embeddings. This more accurately matches the learning the actual actor has to do to extract the information from the nodes as well as the evaluation of the alternative embeddings. The accuracy of the reconstruction was then used to grade the embeddings.

To ascertain whether the embeddings had successfully learned the graph or just memorized the data two additional decoders which the embeddings had not been trained with were introduced.

- Hop distance decoder: instead of decoding the distance it is trained to reconstruct the number of hops to walk between two nodes on the graph. The length of segments in the layout varies greatly so the number of hops to traverse a path is distinct from the distance. The hops are also counted as if the graph was undirected, meaning that many paths will be significantly shorter. In network architecture it is the same as the distance decoder 2.3.2. The loss is taken as the MSE between the prediction and a `log1p` norm of the number of hops.
- Shared lock decoder: given two input segment embeddings it is tasked with reconstructing the Jaccard similarity between the sets of segments locked by each of the inputs. The loss is taken as the MSE between predicted and actual Jaccard similarity.

4.2.2 Laplacian + community embeddings

For these comparative embeddings the same decoders architectures using the same losses used for training and evaluating the segments in the Decoder trained embeddings were trained on the deterministic embeddings. The accuracy of the reconstruction was then used to grade the embeddings.

5

Task Assignment Network

This chapter presents the design and implementation of the task assignment network. It covers how the environment state is represented, the network architecture, reward function, training procedure, and evaluation steps.

5.1 System Overview

This section begins with a high-level overview of the entire system pipeline to provide the reader with a clear picture of the overall structure. The pipeline starts with the initialisation of the environment, loading in the selection environment, placing agvs and tasks. It then sends a first state observation to the actor. The actor can be any function or heuristics that is capable of mapping an observation into a valid task assignment action. The actor then sends back the action, and the environment executes that action. The environment then sends a new observation and the loop repeats. This run loop repeats indefinitely or until a specific termination condition is met. Figure 5.1 shows how this is handle.

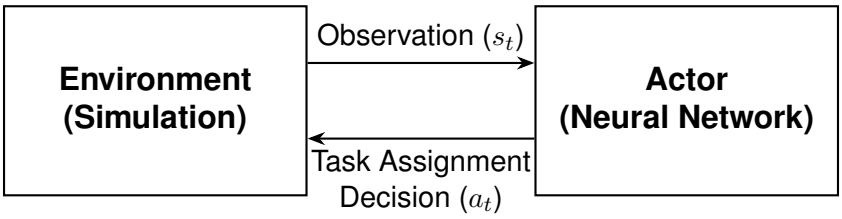


Figure 5.1: Overview of the system interaction during deployment.

5.2 Observation

The environment constructs an observation that represents a snapshot of the current warehouse state. The observation is structured as a graph $G = (V, E)$, where nodes represent AGVs and tasks, and edges encode the relationships between them. The physical travel time between the two nodes. The environment consists of two fundamentally different entities: AGVs and tasks. These are represented as two

distinct node types, AGV nodes and task nodes, giving a total of $n_{agv} + n_{task}$ nodes per observation graph.

Connecting all AGV nodes to all task nodes creates a fully connected bipartite graph, that contains all relevant and practical information the actor should require.

5.2.1 Node features

The two node types carry different feature vectors, shown in figure 5.2. An AGV node's feature vector consists solely of the pretrained static segment embedding of the segment the AGV currently occupies, encoding its structural position in the warehouse graph. A task node carries more information: the pretrained station embeddings for both the pickup and dropoff stations, encoding the structural context of the task. Additionally, a normalised and inverted priority score is included, where priority 1 is the most important task. Finally, the normalised waiting time of the task is included, so the network can identify tasks that have been waiting longest for assignment.

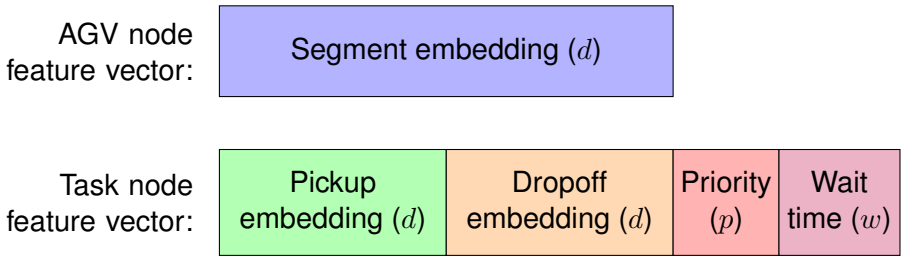


Figure 5.2: Node feature vectors for AGV nodes and task nodes. d is the dimension of the pretrained static embeddings, p is the normalised priority and w is the normalised waiting time.

5.2.2 Heterogeneous observation

An extension of the bipartite observation graph is proposed in which AGV-AGV edges are added alongside the existing task-AGV edges. This introduces awareness of other AGVs directly into the observation, allowing each AGV to incorporate information about the current positions and states of its peers when forming its embedding. The resulting graph is no longer bipartite, as edges now exist within the AGV node set. The motivation for this extension is that coordinated task assignment may benefit from explicit inter-AGV communication, particularly in dense traffic scenarios where the assignments given to one AGV affect the congestion experienced by others.

5.3 Network Architecture

The task assignment networks (TAN) job is to map a state observation into a task assignment action. As seen in figure 5.1 the TAN is the right box receiving the observation and returning an action. To do this the task assignment network consists of three sequential components: a feature projection layer, a GNN backbone, and a decision head that outputs a policy and value estimate. A visual overview of the entire network architecture can been seen in figure 5.3, clearly showing the split decision head.

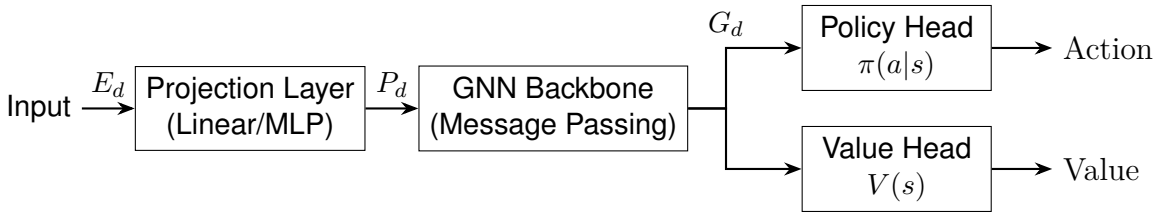


Figure 5.3: Visualisation of the task assignment network architecture. The input graph is first projected to a common dimension, processed by the GNN backbone through message passing, and then passed to the policy head for action selection and the value head for state value estimation.

The three components are interchangeable, as each module operates independently. The only constraint is that the output dimension of each component must match the input dimension of the next: the projection layer outputs P_d, which the GNN backbone expects as input, and the GNN outputs G_d, which the decision head expects.

5.3.1 Feature Projection

The feature projection layer is the first component of the network, responsible for mapping the raw node features into a common embedding space. Since AGV nodes and task nodes have different input dimensions E_d^{agv} and E_d^{task} respectively. A shared GNN backbone cannot be applied directly. The projection layer solves this by applying an independent linear transformation to each node type.

$$x_i^{proj} = \mathbf{W}x_i + \mathbf{b} \tag{5.1}$$

In equation 5.1 $\mathbf{W}$ and $\mathbf{b}$ are learnable weight matrix and bias vector. After projection, all nodes have the same dimension P_d, regardless of their original feature size. This allows the GNN backbone to process all nodes uniformly. The projection can be either a single linear layer or a multi-layer perceptron, depending on how much non-linearity is desired in the feature transformation.

In this implementation, the projection layer is a single linear layer projecting the node features from their respective input dimensions to P_d.

5.3.2 Shared GNN architecture

All GNN implementations follow the same design philosophy: they are modular and interchangeable within the pipeline, allowing each architecture to be swapped out with minimal effort. The number of message passing layers is also configurable across all implementations, allowing the depth of the GNN to be adjusted without architectural changes.

All implementations share the same objective: aggregate information from the graph via message passing and produce updated AGV node embeddings. Task node features remain static throughout, serving solely as a source of information for the AGV nodes. The architectures differ in the mathematical formulation of the message passing and update steps.

5.3.3 Implementation of GCN

The GCN framework follows the message passing formulation described in section 2.4.1, with the specific implementation details described here. The implementation follows the PyTorch Geometric message passing convention, applying the weight matrix $\mathbf{W}$ to all node features before the message passing begins, and adding the bias $\mathbf{b}$ after aggregation rather than inside the message function, avoiding redundant computations of $\mathbf{W}x_j$ for each edge separately.

Two modifications are made to the standard GCN formulation. First, self-loops are added to the graph prior to message passing, with an edge attribute value of zero, ensuring each node aggregates its own features. Second, the standard GCN does not incorporate edge attributes. To address this, the edge attribute e_{ij} is concatenated with the transformed neighbour feature $\mathbf{W}x_j$ and projected back to the original dimension P_d via a linear layer. Equation 5.2 shows the calculation of each separate message.

$$m_{ij} = \text{norm}_{ij} \cdot \text{Linear}\left([\mathbf{W}x_j | e_{ij}]\right) \tag{5.2}$$

where norm_{ij} is the degree normalisation from equation 2.4. The aggregated message is then summed across all neighbours and the learnable bias $\mathbf{b}$ is added. The implemented update step follows that of the original update function, equation 2.5, so no non-linearity is added to the node feature vectors during the message passing or update step.

5.3.4 Implementation of GAT

The GAT implementation is similar to the mathematical presentation in 2.4.2, with some design choices that are not included in the original theory. Similar to the GCN implementation the weight matrix $\mathbf{W}$ is applied to all node features before the message passing begins, projecting them to the output dimension G_d.

The key difference compared to the GCN is the use of learned attention coefficients

rather than the fixed degree normalisation that GCN uses. The attention score between node i and node j is computed by concatenating the transformed features of both nodes along with the edge attribute, and passing them through a learnable attention vector $\mathbf{a}$.

$$e_{ij} = \text{LeakyReLU}_{0.2}\left(\mathbf{a}^T[\mathbf{W}x_i \| \mathbf{W}x_j \| e_{ij}]\right) \tag{5.3}$$

This extends the standard formulation from equation 2.6 by incorporating the edge attributes directly into the attention score computation, allowing the network to weight neighbours based on both node features and travel cost. The attention scores are then normalised across each node's neighbourhood using the SoftMax function. The final aggregated message is then a weighted sum of transformed neighbour features.

Self-loops are added prior to message passing to also aggregate the node's feature vector, consistent with the GCN implementation. In the update step ELU activation function is applied to the aggregated message, introducing non-linearity to the new feature vector, as defined in the original update function 2.7.

5.3.5 Implementation of GT

The GT follows the transformer style message passing formulation described in section 2.4.3, with specific implementation details described here. Unlike the GCN and GAT implementations, the GT does not incorporate edge attributes in the message passing step. Instead, the attention operates purely on the node features.

The implementation uses multi-head attention, where each head independently computes query, key, and value vectors of dimension d_{head}. LayerNorm is applied to the input features before the attention computation:

$$\mathbf{q}_i = \mathbf{W}_Q\text{LayerNorm}(x_i), \mathbf{k}_j = \mathbf{W}_K\text{LayerNorm}(x_j), \mathbf{v}_j = \mathbf{W}_V\text{LayerNorm}(x_j) \tag{5.4}$$

The attention scores are computed as scaled dot-product attention, equation 2.8, and normalised using a sparse SoftMax across each node's neighbourhood.

The update step follows the original transformer formulation from equations 2.9 and 2.10. A skip connection projects the input to the output dimension. The projected input is then summed with the aggregated messages. The sum is then normalised and fed through a simple feed forward network that consists of two linear layers with a gaussian error linear unit activation function in between. The output from the feed forward network (FFN) is then summed with a skip connection around the FFN to create the final output embeddings.

The GT supports configurable number of layers, attention heads and optional concat flag. If the concat flag is set to true the output dimension becomes $n_{heads} \times d_{head}$, otherwise the heads are averaged to maintain dimension G_d. During training, dropout is applied in the attention and feed-forward layers to reduce overfitting.

5.3.6 Policy Head

The policy head maps the node embeddings produced by the GNN block to a probability distribution over task assignments. The architecture is inspired by the attention mechanism used in transformers. Query vectors are computed from the AGV node embeddings, while key vectors are computed from the task node embeddings.

The task logits are then computed using scaled dot-product attention between the AGV queries and task keys. The resulting logits define the policy distribution used for task assignment selection.

5.3.7 Value Head

The value head estimates the value of the current graph state. To obtain a global graph representation, the AGV node embeddings and task node embeddings are pooled separately.

The pooled embeddings are then concatenated and normalised to form a global state representation. This representation is passed through a two-layer multilayer perceptron with a ReLU activation function to produce a scalar value estimate representing the value of the graph observation.

5.3.8 Heterogeneous network extensions

All three GNN architectures have a heterogeneous counterpart: GATH, GCNH, and GTH, that incorporate the AGV-AGV edges described in Section 5.2.2. Each heterogeneous architecture maintains two separate message passing paths that run in parallel: one operating on the task-AGV edges and one operating on the AGV-AGV edges. The two paths are identical in their mathematical formulation but are fully independent, each with its own weight matrices and learnable parameters. Task node features remain static throughout the message passing process. AGV nodes receive messages from both paths, which are subsequently combined via an aggregation function.

The task-AGV edges are constructed identically to the homogeneous case, with edge attributes encoding the travel cost from the cost matrix. The AGV-AGV graph is fully connected, with every AGV connected to every other AGV including itself. As no cost signal is associated with AGV-AGV connections, the corresponding edge attributes are set to zero, meaning the AGV-AGV path operates purely on node features.

Four aggregation strategies are supported for combining the two message streams: sum, mean, max, and a learned sigmoid gate. The gated aggregation is the most expressive of the four, computing a per-feature weighting vector $\mathbf{g} \in [0,1]^{G_d}$ from the concatenation of the two message streams via a linear layer followed by a sigmoid activation. The final AGV embedding is then computed as:

$$\mathbf{h}_i = \mathbf{g} \odot \mathbf{h}_i^{\text{task}} + (1 - \mathbf{g}) \odot \mathbf{h}_i^{\text{AGV}} \tag{5.5}$$

where $\mathbf{h}_i^{\text{task}}$ and $\mathbf{h}_i^{\text{AGV}}$ are the aggregated messages from the task-AGV and AGV-AGV paths respectively, $\odot$ denotes element-wise multiplication. The remainder of the pipeline is identical to the homogeneous architectures.

5.4 Reward Function

The reward function is important as it describes the goal the actor should strive after. The current reward function has two main parts: positive and negative rewards. The positive rewards is given for each assigned task and each successfully completed task. The reward is scaled with the tasks priority, so that tasks with higher priority give more reward.

A punishment is given for neglecting tasks, for each task that is left without any AGV assigned to it for a set amount of time over the average waiting time, is given a small penalty scaled with priority. The individual task a are compared to the average waiting time so that if all tasks are waiting they are not punished. An extreme penalty is given to the actor if the system deadlocks.

5.5 Training

To train the task assignment network, the PPO algorithm presented in section 2.5.2 was used. PPO was chosen for its sample efficiency, reusing each rollout buffer for multiple gradient update steps, and for its stable training behaviour through the clipped surrogate objective. Each training run consists of a fixed number of iterations, where each iteration follows the three phases of PPO: rollout, advantage calculation, and update

During the rollout phase the current policy is used to gather T sequential decision steps from the environment without gradient computation. For each step the model produces a policy and a value estimate, and a task assignment is sampled. The action is produced sequential for each AGV. After an AGV is assigned a task, that task and AGV is masked from the logits, by setting the row and column to negative infinity, before the next AGV is assigned a task. Ensuring that the AGV and task are not used for two assignments during one step.

After the model has returned an action to the simulation the environment then steps forward and returns the next observation and reward. If the environment should deadlock during the rollout, this is detected and the environment is reset. The rollout stores observations, actions, log probabilities, value estimates, reward and timing information.

In the next phase the GAE advantages and returns are computed in a backwards pass over the rollout buffer, following equations 2.15 and 2.18. Both advantages and returns are normalised to zero mean and unit variance. Advantage normalisation stabilises the policy gradient and is commonly done for PPO implementations, while return normalisation reduces the target scale for the value function, which is

important given that raw returns can reach into the thousands depending on the reward function.

During the update phase, the rollout buffer is randomly shuffled and split into mini-batches of size B. For each mini-batch the model is evaluated on the stored observation to produce new log probabilities and value estimates. The PPO clipped surrogate loss, value loss and entropy bonus are then calculated according to equation 2.19. The value loss is additionally clipped, in a similar way to the surrogate loss, to prevent the value function form changing too rapidly in a single update. All gradients are then clipped to a maximum norm of 0.5 before the optimiser step. The update step is then repeated for E number of epochs per iteration. So the total number of gradient steps during the training is then: $n_{iterations} \times E \times \frac{T}{B}$

All three GNN variants are trained with the same hyperparameters, shown in table 5.1. This is done to make it easier to compare the variants, even though there is a high chance that different architectures require different hyperparameters.

Table 5.1: Training hyperparameters used for PPO

Hyperparameter	**Value**
Rollout length, T	512
Epochs per iteration, E	5
Mini-batch size, B	64
Learning rate	3×10^{-4}
Discount factor, γ	0.999
GAE, λ	0.95
Clip, ϵ	0.2
Value loss coefficient, c_1	0.5
Entropy coefficient, c_2	0.01
Gradient clip norm	0.5

5.6 Evaluation

To evaluate the trained actor models, two methods were developed. During training all relevant information is saved and stored alongside the checkpoints. This makes it easy to track if some checkpoints performed better than the others. The data includes but is not limited to: average reward, entropy, policy loss, value loss, total loss, and deadlocks.

The idea is that these metrics together should give a clear picture whether a model actor has been able to actively learn a good policy. However, these metrics are not enough. Therefore, an evaluation script that tests the model actors performance were developed.

The evaluation method used was to do simulation runs with the trained model as the actor. The model runs for a set amount of decision steps in the environment, for multiple episodes. This is done to ensure that outlier episodes do not give faulty

information about the model and degrade the evaluation. While this could give a good picture of how the model performed, it was still required to have something to compare it with.

Therefore, three baseline heuristics were used to compare the results with. The first heuristic was a pure greedy actor. It assigned tasks purely based on the distance between the AGVs and tasks. It creates a long list of all possible AGV-task pairs, and then goes through that list assigning the AGV-task pair with the least travel cost, and masking it so no other AGV can be assigned that task.

The second, much simpler heuristics is a random actor. Each time it is to assign tasks, it generates a random combination of AGV-task pairs, and then uses those pairs for task assignment. The final baseline is derived from the heuristic currently deployed by MAXAGV in production. It extends the greedy assignment principle with a number of additional conditions that improve upon the basic greedy strategy. As the specific implementation details are proprietary, they cannot be disclosed. Nevertheless, this actor serves as a meaningful reference point for the performance achievable by a well-tuned, domain-specific heuristic in this environment.

When evaluating these three heuristics together with an untrained and trained version of the model are run. During the evaluation, statistics are gathered for each actor covering reward-based metrics: average, minimum, maximum, and standard deviation of the reward. Operational metrics are also recorded, including task pick-ups, task completions, throughput, and number of deadlocks. These statistics give a clear insight into how the model performs and how it has learned to behave.

6

Results

6.1 Embeddings

The method used for grading the embeddigns was to see how well they were able to reconstruct various relevant metrics regarding the segments in the layout they were meant to represent. To this method a number of decoders were trained on the embedding vectors. As a reminder, these were:

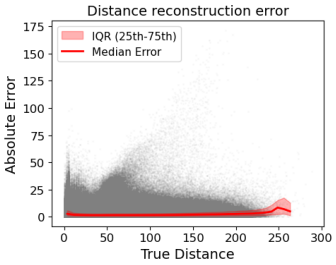
- Distance: a 128,64,1 MLP was trained to reconstruct distance between the segments represented by 2 concatenated embedding vectors.
- Lock relation: multiplies 2 embedding vectors element wise and uses an SLP to predict whether or not they share a lock relation.
- On path: a 192,64,1 MLP was trained to classify whether the middle of 3 concatenated embedding vectors corresponds to a segment which is on the shortest path between the two other segments.
- Station distance: Same as distance but one of the destination embedding is instead the embedding of a station.
- Station on path: Same as on path but the destination embedding corresponds to a station rather than a segment

Degree was omitted from these metrics. The in and out degrees of all segments is much less total information than all other metrics used meaning that the embeddings can store them fully rather than needing to be compressive and encode patterns as with all other metrics used. For all decoders the degree was reconstructed to very high accuracy. When computing the evaluation metrics for the decoders 4 000 000 samples of node pairs were randomly chosen for each of the metrics. This is low relative to the total number of possible node pairs, but it was deemed sufficient to judge embedding performance.

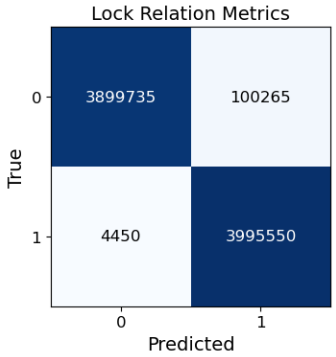
6.1.1 Decoder trained embeddings

For the decoder trained embeddings both the performance of the decoders used in training the embeddings as well as decoders trained from scratch on frozen embeddings are used. This is to demonstrate that the embeddings formed during training

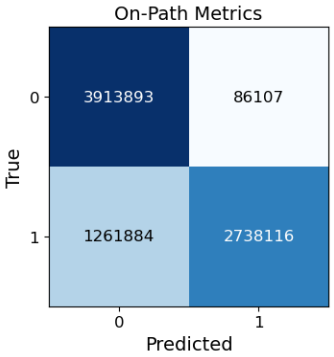
have encoded the information such that it is available to any network trained on the embeddings rather than just the decoders used during training the embeddings.



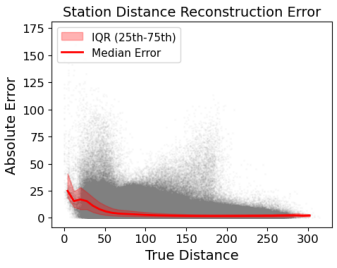
(a) Distance reconstruction error.



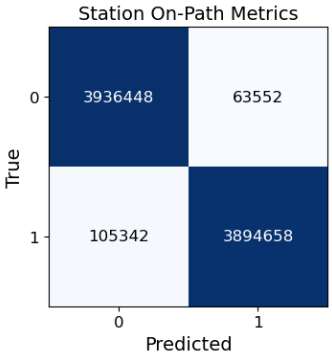
(b) Lock relations metrics.



(c) On-path classification metrics.

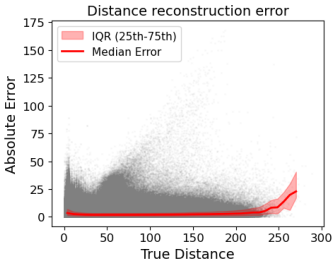


(d) Station distance error.

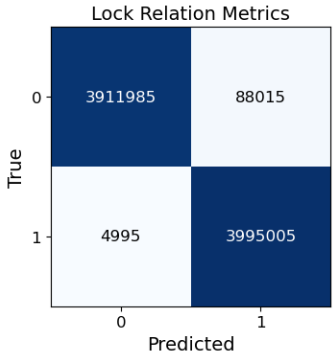


(e) Station degree distributions.

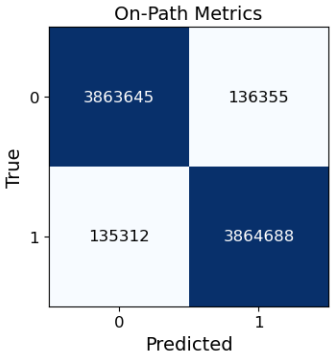
Figure 6.1: Performance metrics for the decoders used in training the embeddings. The distance reconstruction plots show a gray scatterplot of all the sampled distance reconstruction errors, as well as the median absolute error in red. The red area is the IQR within which 50% of the reconstruction errors are. The IQR is very thin for most of the range of distances, indicating that most reconstruction errors fall very close to the median. The worsening of performance at very low and very high distances can be explained by the fact that very few datapoints exist at those extremes.



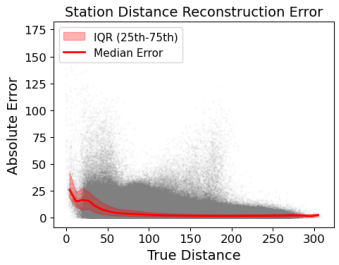
(a) Distance reconstruction error.



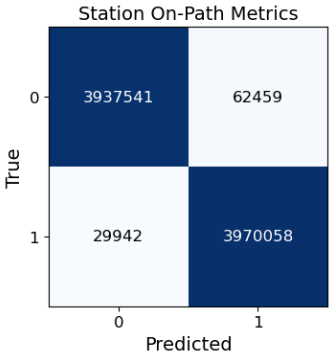
(b) Lock relations metrics.



(c) On-path classification metrics.

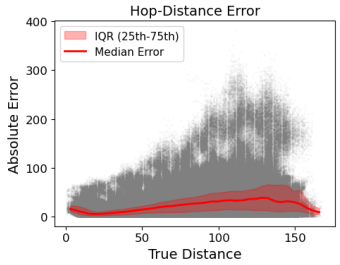


(d) Station distance error.

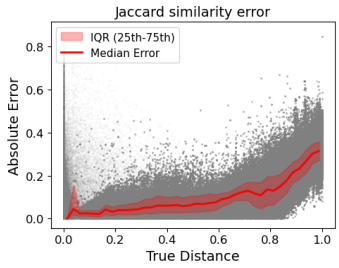


(e) Station degree distributions.

Figure 6.2: Reconstruction metrics for the decoders trained from scratch on the frozen embeddings. The distance reconstruction plots show a gray scatterplot of all the sampled distance reconstruction errors, as well as the median absolute error in red. The red area is the IQR within which 50% of the reconstruction errors are. The IQR is very thin for most of the range of distances, indicating that most reconstruction errors fall very close to the median. The worsening of performance at very low and very high distances can be exaplained by the fact that very few datapoints exist at those extremes.



(a) Graph step distance recontruction error as function of target steps.



(b) Jaccard similarity between lock relations present in node pairs. Reconstruction error plotted as function of target Jaccard similarity.

The reconstruction error is significant for the the hop distance, indicating that the embeddings have encoded distance in terms of time rather than number of segments. The Jaccard similarity reconstruction is quite good for most of 0 to 1 range. The quality seems to significantly drop of from 0.8 to 1. It's not for lack of data, as there are plenty of datapoints in that range of values.

The reconstruction error is significant for the the hop distance, indicating that the embeddings have encoded distance in terms of time rather than number of segments. The Jaccard similarity reconstruction is quite good for most of 0 to 1 range. The quality seems to significantly drop of from 0.8 to 1. It's not for lack of data, as there are plenty of datapoints in that range of values.

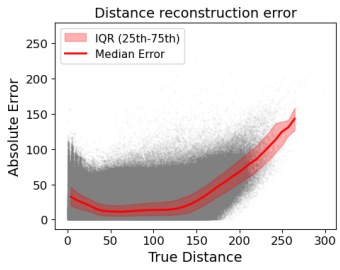
Figure 6.3: Reconstruction metrics for the decoders trained from scratch on the frozen embeddings to reconstruct metrics that the embeddings were not originally explicitly trained on. The reconstruction plots show a gray scatterplot of all the sampled absolute reconstruction errors, as well as the median absolute error in red. The red area is the IQR within which 50% of the reconstruction errors are. The IQR is significantly wider than the metrics for which the embeddings were explicitly trained.

The reconstruction error is significant for the the hop distance, indicating that the embeddings have encoded distance in terms of time rather than number of segments. The Jaccard similarity reconstruction is quite good for most of 0 to 1 range. The quality seems to significantly drop of from 0.8 to 1. It's not for lack of data, as there are plenty of datapoints in that range of values.

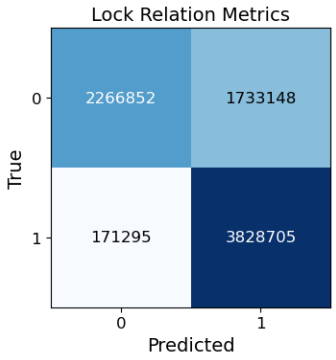
Both sets of decoders perform very well, the only exception being the on path decoder used to train the embeddings. Despite it's poor performance the information seems to have ended up being represented in the embeddings anyway as the retrained decoder performed much better on the same embeddings. Both the lock relations and the distances are significantly larger in terms of total shannon entropy than the node embedding vectors. This means that the embeddings need to encode patterns rather than memorizing the information itself, which they seem to have done. The retrained decoders performing better than the originals is most likely

due to more stable training when only updating the decoders one at a time rather than simultaneously updating the decoders and embeddings.

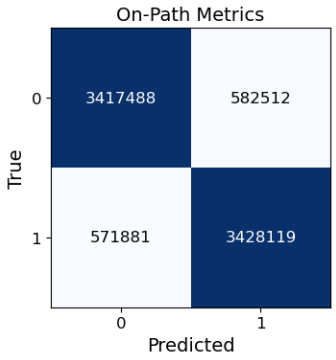
6.1.2 Laplacian + community embeddings



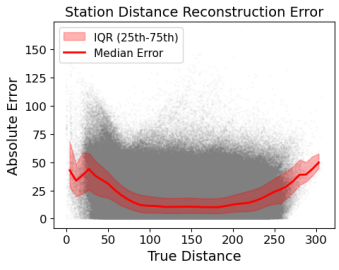
(a) Distance reconstruction error.



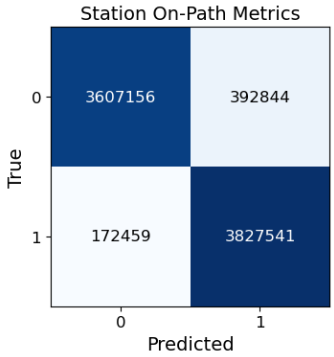
(b) Lock relations metrics.



(c) On-path classification metrics.



(d) Station distance error.



(e) Station degree distributions.

Figure 6.4: Performance metrics for the decoders trained on Laplacian + comunity embeddings.

The metrics for the decoders trained on the embeddigns formed of concatenated laplacian and community components performs significantly worse across all metrics with the sole exception of identifying wether or not a point if on a path between two others, at which it performs similarly well to the trained embeddings. While the distance reconstruction performance is significantly worse than the learned embeddings it is still decent. Especially in the region where most distances lie.

6.2 Task assignment network

The task assignment network was trained using the PPO algorithm, described in Section 2.5.2. To ensure a fair comparison between architectures, all variants were trained with identical hyperparameters for 500 iterations, using the same reward

function throughout. The trained models are subsequently evaluated both on their training behaviour and on their performance relative to one another and to conventional heuristic baselines. This section first examines the training metrics, followed by a comparative evaluation of task assignment performance.

6.2.1 Training performance

To evaluate and compare the training behaviour of the six architectures, training metrics were logged throughout the training process. These models were all trained with the reward function described in section 5.4.

The raw training metrics for the GAT model are shown in figure 6.5 as a representative example of the per-iteration signal noise present in the unsmoothed data. The raw metrics for all six architectures, including GAT, are collected in Appendix A for ease of comparison. The mean reward per step shows high variance across iterations, making it difficult to draw conclusions from the raw signal alone. To facilitate comparison across architectures, EMA-smoothed curves are used in all subsequent figures.

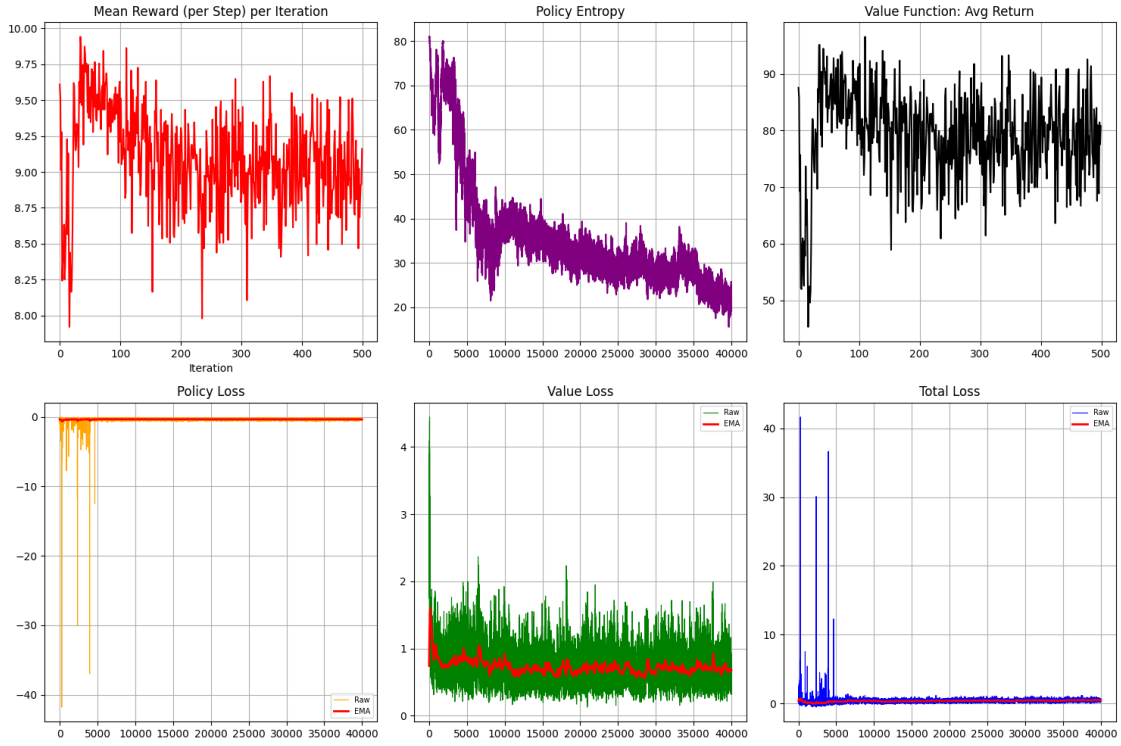


Figure 6.5: Raw training metrics for model using the GAT architecture for the GNN body of the task assignment network. The different values for the x-axis arises from the fact that some data is gathered once per iteration and some once per update. As these models were trained for 5 epochs and 16 batches, each iteration does 80 steps. This corresponds to an x-axis ratio of 500 iterations to 40,000 update steps.

Figure 6.6 presents EMA-smoothed training curves for all six architectures. In

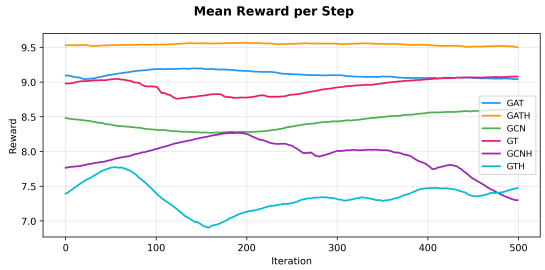
contrast to the previous training runs, the mean reward curves show remarkably little variation over the course of training. All architectures operate within a narrow band between approximately 7.3 and 9.5, with no architecture exhibiting a clear upward trend. GATH maintains the highest mean reward throughout training, remaining flat near 9.5. GAT is stable just below at approximately 9.1. GT dips moderately during mid-training before recovering to approximately 9.1 by the end. GCN shows a slow, gradual increase from around 8.5. GCNH rises through the first half of training before declining in the second half, ending near 7.3. GTH exhibits the most pronounced dip, falling to approximately 7.0 around iteration 150 before partially recovering to 7.5. The absence of strong upward trends across all architectures suggests that the reward signal, while producing stable dynamics, does not drive substantial policy improvement over the training horizon.

The entropy curves reveal markedly different exploration dynamics across architectures, with a clear divide between those that maintain exploration and those that collapse. GATH maintains the highest entropy throughout training, remaining near 70–75 nats with only minor fluctuations, indicating sustained broad exploration. GAT begins at a similarly high level but undergoes a gradual collapse over the course of training, ending near 20 nats by iteration 500. GCN starts at very low entropy and slowly recovers before a sharp drop back to where it started. GT collapse early to near zero and remain there, indicating rapid premature convergence to near-deterministic policies. GCNH undergoes a sharp entropy collapse around update step 10 000, after which it also operates near zero. GTH collapses early but partially recovers, subsequently oscillating between approximately 5 and 15 nats for the remainder of training, suggesting recurring instability in the policy distribution.

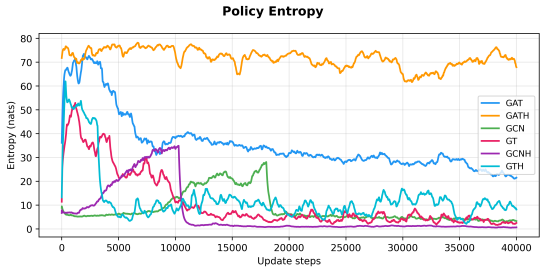
The policy loss curves are largely uniform across architectures, with all models clustering tightly just above -0.4 throughout training. GATH is the notable exception, exhibiting recurring sharp downward spikes that indicate episodic large policy updates. GCNH shows one large spike early in training before joining the cluster. The general uniformity of policy loss across architectures that differ substantially in their entropy behaviour suggests that the PPO clipping mechanism is functioning as intended, preventing catastrophically large updates even when entropy collapses.

The value loss curves separate the architectures into three groups. GAT, GATH, and GTH maintain low and relatively stable value losses throughout training, converging to the 0.6–0.8 range. GCN and GT show moderate losses with occasional spikes but remain broadly stable. GCNH is the clear outlier: after a stable early phase, the value loss diverges substantially from approximately update step 30 000 onward, oscillating between 1.5 and 2.5 without recovering. This late-stage instability in the value function coincides with the continued rise in deadlocks seen in figure 6.7, and suggests that GCNH's policy updates become increasingly poorly calibrated in the second half of training.

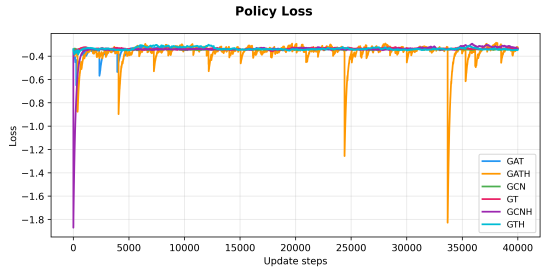
The number of deadlocks per iteration is shown in Figure 6.7. The architectures exhibit clearly distinct deadlock profiles. GATH achieves the lowest and most stable deadlock count throughout training, remaining near 4–5 deadlocks per iteration with no significant trend. GAT starts with an initial spike to approximately 16, drops to



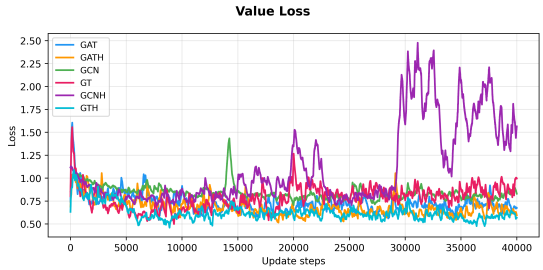
(a) Mean Reward per Step



(b) Policy Entropy



(c) Policy Loss

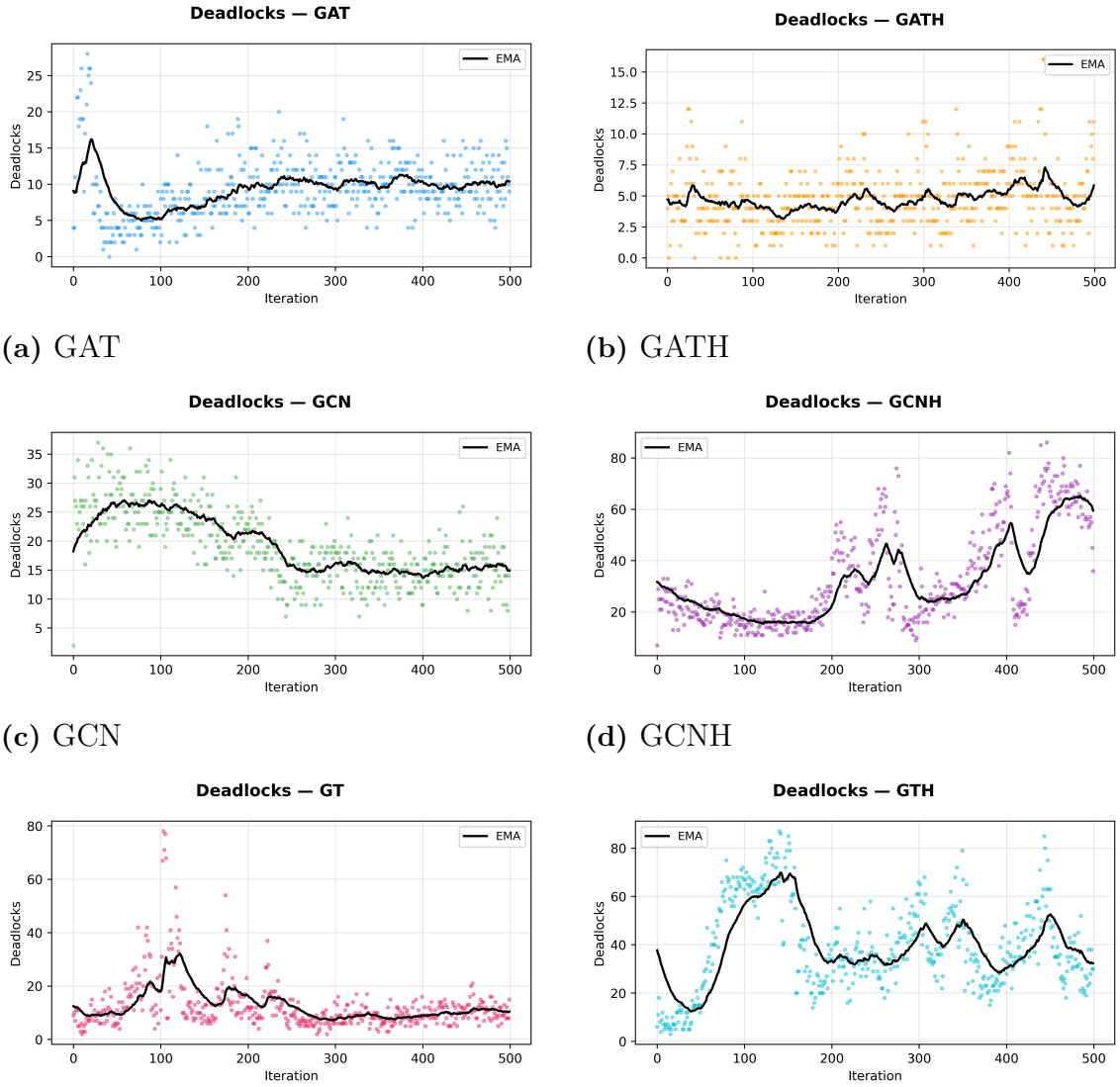


(d) Value Loss

Figure 6.6: EMA-smoothed training curves for all six architectures. Note the differing x-axis scales: mean reward is logged per iteration, while entropy, policy loss, and value loss are logged per update step.

around 5 by iteration 75, then gradually rises and plateaus near 10 for the remainder of training. GCN begins at approximately 18, rises to a peak near 27 around iteration 100, then shows a sustained and consistent decline to approximately 15 by iteration 500, suggesting the policy progressively learns to reduce conflicting assignments despite its collapsed entropy. GT exhibits a sharp spike to approximately 30 around iteration 100 followed by a rapid decline, stabilising near 10 for the second half of training.

GCNH begins near 25, declines modestly to around 17 by iteration 150, but then rises sharply in two successive waves, reaching approximately 65 deadlocks per iteration by the end of training. GTH displays the most volatile profile: starting near 35, it briefly drops before spiking to approximately 68 around iteration 130, then oscillates with recurring peaks between 30 and 50 for the remainder of training. The effect of adding heterogeneous AGV–AGV edges is therefore highly architecture dependent. GATH stands out as a clear beneficiary, maintaining the lowest deadlock count of all six models throughout training. GCNH and GTH, by contrast, exhibit consistently elevated and worsening deadlock counts relative to their homogeneous counterparts, suggesting that the additional graph complexity introduced by AGV–AGV edges creates optimisation difficulty that the current training configuration is unable to resolve for these architectures.



(a) GAT **(b)** GATH

(c) GCN **(d)** GCNH

(e) GT **(f)** GTH

Figure 6.7: Number of deadlocks per iteration with an EMA for clarity. Notice the different scales on the y-axis, as the y-axes are not shared across subplots. Left column: homogeneous models. Right column: heterogeneous models.

6.2.2 Evaluation

The evaluation results are presented in table 6.1. The primary metric of interest is throughput, defined as the number of completed tasks per episode divided by the total simulated episode time. Throughput captures task assignment efficiency independently of episode length. Mean reward and completed tasks are included as supporting metrics, while average steps and deadlock rate characterise episode termination behaviour.

The results present a mixed picture with respect to the effect of training, some architectures show improvement over their untrained baseline. Trained GAT achieves a throughput of 0.0159 ± 0.0044 compared to 0.0148 ± 0.0050 for untrained GAT, and trained GT improves from 0.0081 ± 0.0025 to 0.0111 ± 0.0025, the largest absolute gain among all architectures. By contrast, GCN degrades sharply under training, from 0.0174 ± 0.0055 untrained to 0.0043 ± 0.0029 trained, the lowest throughput of any model variant. GATH and GTH show negligible change between trained and untrained. The inconsistency across architectures suggests that the training pipeline is capable of producing improvement in some settings, but does not do so reliably.

All trained and untrained model variants fall substantially short of the heuristic baselines. Greedy achieves the highest throughput at 0.0328 ± 0.0082, followed by MAXAGV at 0.0273 ± 0.0089 and Random at 0.0218 ± 0.0100. The best-performing model variant, untrained GCN, reaches 0.0174 ± 0.0055, roughly half of Greedy's throughput. The performance gap is also reflected in average episode length: Greedy episodes last 151.6 steps on average, while untrained GCN lasts 109.8 steps and trained GCN terminates after only 12.3 steps.

A notable finding is that the heterogeneous graph variants do not show a systematic advantage over their homogeneous counterparts. In the homogeneous group, untrained GCN (0.0174) and trained GAT (0.0159) both outperform the best heterogeneous variant, untrained GCNH (0.0153). The GATH variants perform among the weakest overall, with throughputs of 0.0069–0.0072, below the homogeneous GT variants. This contradicts the expectation that heterogeneous AGV–AGV edges would provide a structural advantage, and suggests that the additional expressiveness of the heterogeneous graph either introduces optimisation difficulty or is not effectively exploited by the policy under the current training configuration.

The near-universal deadlock rate across all actors remains a characteristic of the evaluation environment. All model variants deadlock in every episode, and even Greedy reaches a deadlock rate of 100%. Only MAXAGV and untrained GCN achieve rates below 100%, at 96.71% and 99.73% respectively. As before, deadlock rate is not a discriminating metric between actors, and the evaluation relies primarily on throughput and average episode length to characterise performance differences.

Table 6.1: Evaluation metrics for all actors, reported as mean ± standard deviation across 365 episodes of at most 500 decision steps each, with 18 AGVs and a task buffer of 100. Episodes terminate upon deadlock or upon reaching 500 steps. Actors are grouped into baseline heuristics and GNN model variants, with each architecture's untrained version included for reference.

Actor	Mean Reward	Completed Tasks	Throughput	Avg. Steps	Deadlock Rate
Random	882.97 ± 745.89	43.15 ± 37.08	0.0218 ± 0.0100	90.8	100%
Greedy	1499.27 ± 816.86	72.93 ± 41.79	0.0328 ± 0.0082	151.6	100%
MAXAGV	1298.89 ± 1057.26	63.91 ± 52.70	0.0273 ± 0.0089	130.7	96.71%
GAT (untrained)	794.19 ± 590.86	38.88 ± 29.55	0.0148 ± 0.0050	82.8	100%
GAT	1132.81 ± 736.11	55.71 ± 36.86	0.0159 ± 0.0044	116.4	100%
GATH (untrained)	279.62 ± 214.46	13.41 ± 10.66	0.0069 ± 0.0021	32.7	100%
GATH	286.85 ± 184.74	13.79 ± 9.18	0.0072 ± 0.0020	33.2	100%
GCN (untrained)	1067.03 ± 825.54	52.37 ± 41.25	0.0174 ± 0.0055	109.8	99.73%
GCN	87.79 ± 108.41	3.89 ± 5.35	0.0043 ± 0.0029	12.3	100%
GCNH (untrained)	806.92 ± 559.07	39.38 ± 28.04	0.0153 ± 0.0046	84.4	100%
GCNH	726.01 ± 625.90	35.25 ± 31.51	0.0127 ± 0.0053	76.5	100%
GT (untrained)	259.47 ± 170.63	12.46 ± 8.46	0.0081 ± 0.0025	29.5	100%
GT	633.73 ± 431.86	31.00 ± 21.61	0.0111 ± 0.0025	66.8	100%
GTH (untrained)	321.85 ± 311.59	15.58 ± 15.49	0.0077 ± 0.0031	35.6	100%
GTH	301.01 ± 292.76	14.55 ± 14.56	0.0076 ± 0.0033	33.5	100%

7

Discussion

7.1 Embeddings

Creating effective embeddings for the segments and stations in the layout used came with some challenges, mostly because the atypical nature of the layout graph as compared to the graphs more commonly used when creating node embeddings, such as social networks or citation graphs. These graphs usually exhibit small world properties. An embedding that is representative of a node can be formed just from its neighbourhood. Contrast that to our graph where the paths whose distances need to be encoded can be multiple hundreds of hops in length and median degree in connections is 1, meaning that the majority of segments are just parts of chains rather than a dense graph. At the same time condensing multiple segments in a chain into a single segment to get a more dense graph is not acceptable as they still each have unique lock relations which is described by a much more dense graph were long range relations are irrelevant. This means that most methods commonly used for creating node embeddings fail in our use case which is why our methods are a bit more atypical.

Because the graph cannot be structurally simplified, standard embedding frameworks struggle to capture its dual nature. We initially considered Node2Vec, a widely successful transductive embedding method. However, Node2Vec relies heavily on biased random walks to capture local neighbourhood structures and structural equivalence.

When applied to our layout graph, Node2Vec fails for two primary reasons:

The sparsity of connections and long-range dependencies: On the connection edges, a random walk is often restricted to a deterministic 1D line (a single path of segments). Because spatial distances between relevant decision points (such as intersections or stations) are vast, a random walk would require an impractically large window size to capture the long-range relationships necessary to estimate travel times or shortest paths.

The disparity between edge types: If the random walk incorporates the lock edges, the walker almost immediately falls into massive, dense clusters of hundreds of nodes. The sparse, spatial pathing information is instantly drowned out by the sheer volume of lock relations. Consequently, the resulting embeddings fail to meaningfully encode

either the spatial geometry or the lock dependencies.

There are also many methods for creating node embeddings that utilize various graph based message passing models. However a message passing network can only "see" as many steps in a graph as far as it is deep in layers. This means that in order to capture the long distance path structure in our graph a message passing based approach would need to perform hundreds of layers of message passing. This would completely destroy the node vectors from over smoothing, all the vectors in the graph would be practically identical.

While the node embeddings we created with out encoding model can be used to train a small network to accurately reconstruct the desired features this is not necessarily an indication that they are a good choice for the input to an RL trained GNN. For example the data might be encoded in a way that makes it to difficult to extract for an RL agent to reliable decode it during training, the learning signal in RL is much more noisy than the gradient optimization used to train the evaluation decoders. The metrics encoded may also not be the best choice for an input. We attempted to choose metrics that would well encapsulate the graph, distances to capture the spatial structure of the graph, whether a node is on the path between two others to capture the long range topology and the lock relations since it was assumed to be important information for a task assignment agent to consider. There is no guarantee that these specific forms for losses are the optimal choice however. For instance, 2 or more step neighbourhoods in the lock relation edges could be the optimal measure to train for.

7.2 Adoption of legacy system

Some hurdles arise when working with reinforcement learning in real commercial systems rather than clean simulations or games as you often see in reinforcement learning examples. Real world AGV systems are often the product of years of iterative development, refinement, and optimization. As such they are often very complex with many special rules and edge cases. Implementing an RL system involves many tasks that depend deeply on the system in the RL model is intended to work in. For example designing the observation space, action space, reward shaping, the simulated representation of the environment all depend deeply on the existing system. A complex existing system then comes with it's own overhead and friction sources when attempting to design an effective RL solution.

If the engineers working on RL are not the same as the developers of the underlyign AGV system another source of friction arises in the form of a knowledge gap between the two teams. As the engineers working on the AGV system are unlikely to have a deep familiarity with what an RL formulation requires, and the RL team lacks a complete picture of the underlying system, both sides end up working from an incomplete understanding of the other's domain. This gap is not simply a matter of communication overhead, its scope tends to scale with the complexity of the system itself. The more rules, edge cases, and domain-specific behaviour the system has, the larger the surface area of things the RL team may or may not need to know,

and the harder it becomes for the system-side engineers to anticipate what is and isn't relevant to convey.

Both these issues are quite likely to arise when adapting an RL solution to a legacy system. AGV systems are not complex out of poor design but necessity, likewise the engineers with a deep understanding of the existing system will likely have more pressing tasks that demand their expertise than to get a deep understanding of RL systems just to communicate information about the legacy system effectively.

7.3 Room for improvement with an RL based task assignment method

Many of the existing studies in multi agent task assignment and pathfinding using RL use grid environments or environments close to grid-like [7][8]. This is structurally very different from most real world storage facilities which are often structured around common high traffic lanes from which AGVs break off into corridors lined with stations wherein tasks are located2.2a. In a grid like, or similarly path dense, environment there is large room for expression in multiagent pathfinding, there are many paths that can be chosen from point a to b giving plenty opportunity to avoid conflicts between individual agents. The same expressiveness in path choice is not reflected in a more realistic layout. Another aspect is that paths from a given starting point to different destinations will vary much more in which segments they include in a grid like layout whereas the sparse, realistic layout is oriented around unidirectional lanes that are, in part, included in the paths to many if not all destinations. This leads to a large degeneracy in the paths that emerge from different task assignment distributions further decreasing the potential for a task assignment algorithm to cleverly avoid conflicts via task assignment. In real world facilities there is also degenaracy in terms of the placements. A common configuration for a storage facility is a dock where payloads are dropped of and another nearby where they are picked up and then a wealth of positions inside a storage facility where they may be stored or from where they may be retrieved. The vast majority of tasks then share that dock, or one of a few different high traffic stations. This is a very efficient layout but it does also contribute to limiting the potential expressiveness of a task assignment algorithm.

7.4 Reward shaping

Reward shaping for an RL agent learning in a complex environment is a very difficult problem. A reward function for a real world AGV system has to optimise for multiple metrics and weigh them carefully. For example a real world AGV system has to be able to:

1. Choose AGV, task pairs in a manner that maximizes throughput.
2. Take care to not leave individual tasks for too long even if choosing them would hamper total throughput. A customer won't be happy with their order

being ignored just because it happened to be in an awkward position in the facility.

3. Different tasks having different priority.

4. Potential facility specific preferences and optimizations.

The different metrics have to be weighed against each other. This weighing is also self dependent. For example, if a certain metric is given a higher priority the process of the simulation will also change with the different weighing of the agent decision, this will in turn impact the contribution of the different reward components to the total reward [22]. One can also include reward contributions which are not directly tied to a desirable metric but rather born from a heuristic to help the agent on the way to reach an optimal solution [23]. A reward function including these sort of components is often referred to as a shaped reward. In this specific thesis one could imagine adding an extra reward when an AGV is assigned to it's closest available task, directly corresponding to the greedy heuristic. Another option would be to add a punishment for AGVs having to wait for one another.

When designing a reward function another obstacle is the potential for the agent to game the reward, learning undesirable behaviour which still leads it to effectively maximize the expected reward [24]. For example in our thesis we tried to add a punishment for leaving singular tasks for longer than the mean in the task buffer. This was intended to teach the agent to not leave singular tasks untouched for extended periods of time. However when this punishment was used during training the AGVs instead learned to deadlock, causing the simulation to be restarted and avoiding the accumulating punishment, which turned out to be greater than the punishment for a deadlock. Not all reward gaming may be this obvious and a more complex reward function invites more opportunities for reward gaming.

Reward shaping can often be entire thesis project in terms of scope [25]. Designing reward functions often requires expert knowledge and hand engineering, and the difficulties are further exacerbated when there are metrics to be optimized by the agent [25]. Designing effective shaping reward functions is challenging and time consuming, requiring multiple iterations of training agents with different shaping rewards, evaluating their performance, and refining accordingly. A process made inefficient because performance measured early in training may be misleading, while performance measured late requires lengthy, computationally expensive training, leading to slow iteration.

The reward functions described here are representative ex maples from the design process. The first, a shaped reward, combined a task pickup and completion signal with a softplus-based waiting penalty. The second, a sparse reward, provided a positive signal only upon task pickup and completion, with no shaping components. The shaped reward produced upward trends in mean reward during training, which initially appeared promising. However, this signal is suspected to reflect the agent learning to optimise the shaping components rather than genuinely improving task assignment behaviour. One such formulation included a dense distance-based signal,

but this inadvertently aligns with the Greedy heuristic, meaning a policy optimising it would not necessarily learn anything beyond what Greedy already captures. Removing the shaping components entirely produced a cleaner signal. Since the reward is event-based, triggering on every task pickup and completion, the sparsity of the sparse reward is partially mitigated in practice, with 18 AGVs and a task buffer of 100, reward events occur regularly throughout the episode. Despite this, no formulation produced reliable improvement over the untrained baseline or the heuristic actors.

7.5 Architecture behaviour

The training metrics present a mixed picture when considered alongside the evaluation results. GATH's sustained high entropy and low deadlock count during training did not translate into competitive evaluation performance, suggesting that broad exploration alone is insufficient if the policy does not converge toward good assignments. On the opposite, GT's early entropy collapse and volatile mid-training deadlock spike did not prevent it from achieving the strongest evaluation improvement over its untrained baseline. GCN presents a striking contradiction: a progressive deadlock reduction and increasing mean reward during training, yet a catastrophic collapse in evaluation throughput from the trained variant. The severe value loss degradation of GCNH during the second half of training offers a possible explanation for why training actively degraded its evaluation throughput relative to the untrained variant.

The difference in training behaviour across architecture are likely partially linked to their respective message passing and update functions, and how these interact with gradient flow during backpropagation. GCN's fixed symmetric normalisation provides no mechanism for the policy to selectively weigh relevant neighbours, which may explain the early convergence. GAT's learned attention weights provide more structured gradient information, potentially explaining its more sustained exploration. The GT architecture, which utilises full self-attention, has considerably more parameters than GAT and GCN. In a case where the reward signal is difficult to specify, this additional complexity may be a liability rather than an advantage. This is consistent with GT's early entropy collapse, and may also explain why the heterogeneous variants GCNH and GTH, which add further complexity through additional edge types and relation-specific message passing, exhibit the most unstable deadlock profiles of all architectures. However, without systematic studies isolating these factors, such explanations remain speculative.

7.6 Reinforcement learning and AI

The results presented in this thesis raise a broader question about the suitability of reinforcement learning as a paradigm for AGV task assignment. While the trained models fail to outperform heuristic baselines in the current evaluation, this does not imply that reinforcement learning is the wrong approach. Rather, it highlights the

conditions under which RL can be expected to offer advantages, and where those advantages may be difficult to realise in practice.

A fundamental challenge is sample efficiency. Training an RL agent requires vast amounts of interaction data before any meaningful policy emerges. For an agent to reach behaviour comparable to Greedy, considerable training is necessary, compared to the trivial implementation of equivalent behaviour as a deterministic heuristic. Heuristics require no training and performs competently from the first step. The computational cost of training six architectures for hundreds of iterations illustrates this disparity clearly. This does not disqualify RL as an approach but it does raise the question of whether the potential performance gains an RL agent may offer over a heuristic justify the cost. The answer likely depends on whether the problem contains structures that a fixed heuristic cannot exploit. In a static, well-defined problem, an optimised heuristic will be difficult to surpass. In a dynamic environment with evolving task distributions and facility configurations, an RL agent may be better positioned to adapt.

The main argument in favour of RL is generalisation. A well-trained policy could in principle adapt to unseen task distributions, varying fleet sizes, and changing facility layouts without modification. Greedy is a fixed rule that makes locally optimal decisions with no capacity to learn from experience or improve with exposure to data. This is particularly relevant in the context of MAXAGV's custom heuristic, where the same system is installed across facilities with differing layout and operational characteristics. A heuristic that performs well in one facility may require manual changes, whereas a good general learned policy could adapt without intervention. If an RL agent were to demonstrate reliable performance across varying facility configurations, the investments in training would become justified. The results of this thesis do not yet demonstrate this capacity, but establishing a working training pipeline and a flexible GNN-based architecture represents a necessary first step toward that goal.

A further consideration is interpretability. Greedy, and heuristic methods in general, are fully transparent. Given an AGV fleet and a set of tasks, the resulting assignment decision is trivially explained in terms of the underlying rules. In contrast, a GNN policy produces assignments through a series of learned transformations that offer no direct insight into the reasoning behind a given decision. In a critical logistics environment where an unexpected assignment could result in a blocked corridor or a missed delivery deadline, the opacity of a learned policy is a concern. Techniques such as attention visualisation may offer a partial insight into the policy's decision making, but the interpretability gap between learned and rule-based methods remains a practical obstacle.

7.7 Future work

The most important direction for future work is the development of a more effective reward function. The reward function must capture the objectives of the system clearly, reinforcing behaviour that moves the policy toward desirable outcomes while

penalising behaviour that moves away from them. As discussed in Section 7.4, this is a non-trivial problem that may require substantial iteration. Throughout this thesis, reward shaping has been identified as the primary suspected cause of the unsatisfactory results, the models receive insufficient signal during training to learn behaviour that meaningfully improves upon the untrained baseline. As a consequence, hyperparameter tuning was deliberately deprioritised. A configuration that produced stable training dynamics was adopted early and maintained throughout, with hyperparameters initialised following the recommendations of the original PPO paper. A more systematic tuning process, conducted separately for each architecture, would likely be warranted once a more informative reward signal is established.

Whether longer training runs would improve performance remains an open question. The current setup collected 256 000 decision steps across 500 iterations, broadly comparable to the 200 000–500 000 training steps reported in [9]. However, the training curves presented in section 6.2.1 show no clear upward trend in any architecture by iteration 500, suggesting that the limiting factor is not training duration but the quality of the training signal. Extending the number of iterations without first addressing the reward specification would therefore be unlikely to yield meaningful improvement.

The node embedding scheme also requires further study. The methods used in evaluating the embeddings are fairly weak and a better pipeline for judging the embeddings would need to be established. This is the first and foremost priority as any improvements to the embeddings cannot be verified to be improvements before an evaluation scheme is in place that is more faithful to the actual training pipeline. One option would be to train an RL agent using different sets of embeddings and judging them by the difference in performance. This comes with the caveat that any other issues with the training pipeline could ruin the results. It would also be exceedingly computationally expensive, likely to the point that it isn't viable, at least not as the sole method of evaluation.

A more complete simulation environment would also benefit future work. In particular, facility-specific deadlock prevention mechanisms would make the evaluation more accurate and bring the setup closer to real deployment conditions. Implementing such mechanisms is non-trivial, as the facility layouts are complex and inherently difficult to make deadlock-free. The layouts are provided by MAXAGV, and while an effort was made to replicate their existing rules, a complete specification was not available, as some rules are facility-dependent and not fully documented.

7.8 Ethics

There are a few aspects to consider in developing a machine learning approach: worker displacement, safety, and energy.

7.8.1 Replacement of human workers with AGVs

One ethical concern of employing AGV solutions rather than human forklift operators is the displaced workforce. This displacement is largely driven by the adoption of AGV technology itself. Any potential efficiency increase from machine learning driven task assignment will most liekly only have negligible second order effects on the displacement of workers through AGV technology becoming more efficient, and therefore, more desirable.

7.8.2 Warehouse safety aspects

It is critical that AGVs operating in proximity to humans be designed with safety in mind. One could imagine that using a machine learning algorithm for controlling AGVs would introduce opacity and stochasticity that might compromise safety and make the software harder to audit or debug. In this case the system that is being replaced is only the task assignment, i.e the assignment of destinations to the individual AGVs. All logic related to navigation and object avoidance will still be handled by deterministic human-written code.

7.8.3 Energy consumption

Machine learning systems have faced criticism for the enormous power draw of the data centres used in training. However, the machine learning models in this project will most likely be very lightweight in comparison, making it trainable on consumer hardware and requiring an negligible amount of power. If the route planning of a fleet of AGVs could reduce total distance travelled it may lead to a marginal reduction in power usage. On the other hand, it is also possible that scheduling optimized for completing tasks as quickly as possible will draw more total energy than current solutions.

Bibliography

[1] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Magazine*, vol. 29, no. 1, pp. 9–20, 2008.

[2] IMARC Group, "Automated guided vehicles market size, share & trends," https://www.imarcgroup.com/automated-guided-vehicles-market, 2025, accessed: 2026-05-19.

[3] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak, "Multi-agent pathfinding: Definitions, variants, and benchmarks," 2019. [Online]. Available: https://arxiv.org/abs/1906.08291

[4] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," 2017. [Online]. Available: https://arxiv.org/abs/1705.10868

[5] F. Popolizio, M. Vinetti, A. Combrink, S. F. Roselli, M. Pia Fanti, and M. Fabian, "Online conflict-free scheduling of fleets of autonomous mobile robots," pp. 3063–3068, 2024.

[6] MAXAGV, "About us," https://maxagv.com/about/, n.d., accessed: 2026-05-12.

[7] J. Gaber, M. Alharbi, D. Gammelli, and G. Zardini, "GRAND: Guidance, rebalancing, and assignment for networked dispatch in multi-agent path finding," 2024.

[8] A. Agrawal, A. S. Bedi, and D. Manocha, "RTAW: An attention inspired reinforcement learning method for multi-robot task allocation in warehouse environments," 2022.

[9] L. Ratnabala, A. Fedoseev, R. Peter, and D. Tsetserukou, "Magnnet: Multi-agent graph neural network-based efficient task allocation for autonomous vehicles with deep reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2502.02311

[10] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks,"

2016. [Online]. Available: https://arxiv.org/abs/1607.00653

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *Proceedings of Workshop at ICLR*, vol. 2013, 01 2013.

[12] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[13] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 1263–1272. [Online]. Available: https://proceedings.mlr.press/v70/gilmer17a.html

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017. [Online]. Available: https://arxiv.org/abs/1609.02907

[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018. [Online]. Available: https://arxiv.org/abs/1710.10903

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf

[18] J. Schulman, F. Wolski, P. D. anda Alec Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[19] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2015. [Online]. Available: https://arxiv.org/abs/1506.02438

[20] F. Foundation, *Gymnasium Documentation*, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666651021000012

[21] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," *CoRR*, vol. abs/1705.07115, 2017. [Online]. Available: http://arxiv.org/abs/1705.07115

[22] A. Kusari and J. P. How, "Predicting optimal value functions by interpolating reward functions in scalarized multi-objective reinforcement learning," 2020.

[Online]. Available: https://arxiv.org/abs/1909.05004

[23] B. Badnava, M. Esmaeili, N. Mozayani, and P. Zarkesh-Ha, "A new potential-based reward shaping for reinforcement learning agent," 2023. [Online]. Available: https://arxiv.org/abs/1902.06239

[24] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," 2016. [Online]. Available: https://arxiv.org/abs/1606.06565

[25] R. Devidze, "Reward design for reinforcement learning agents," 2025. [Online]. Available: https://arxiv.org/abs/2503.21949

A

Raw training metrics

The following figures present the raw, unsmoothed training metrics for all four architectures in landscape format for legibility. These are provided as a reference for the EMA-smoothed curves presented in Figure 6.6.

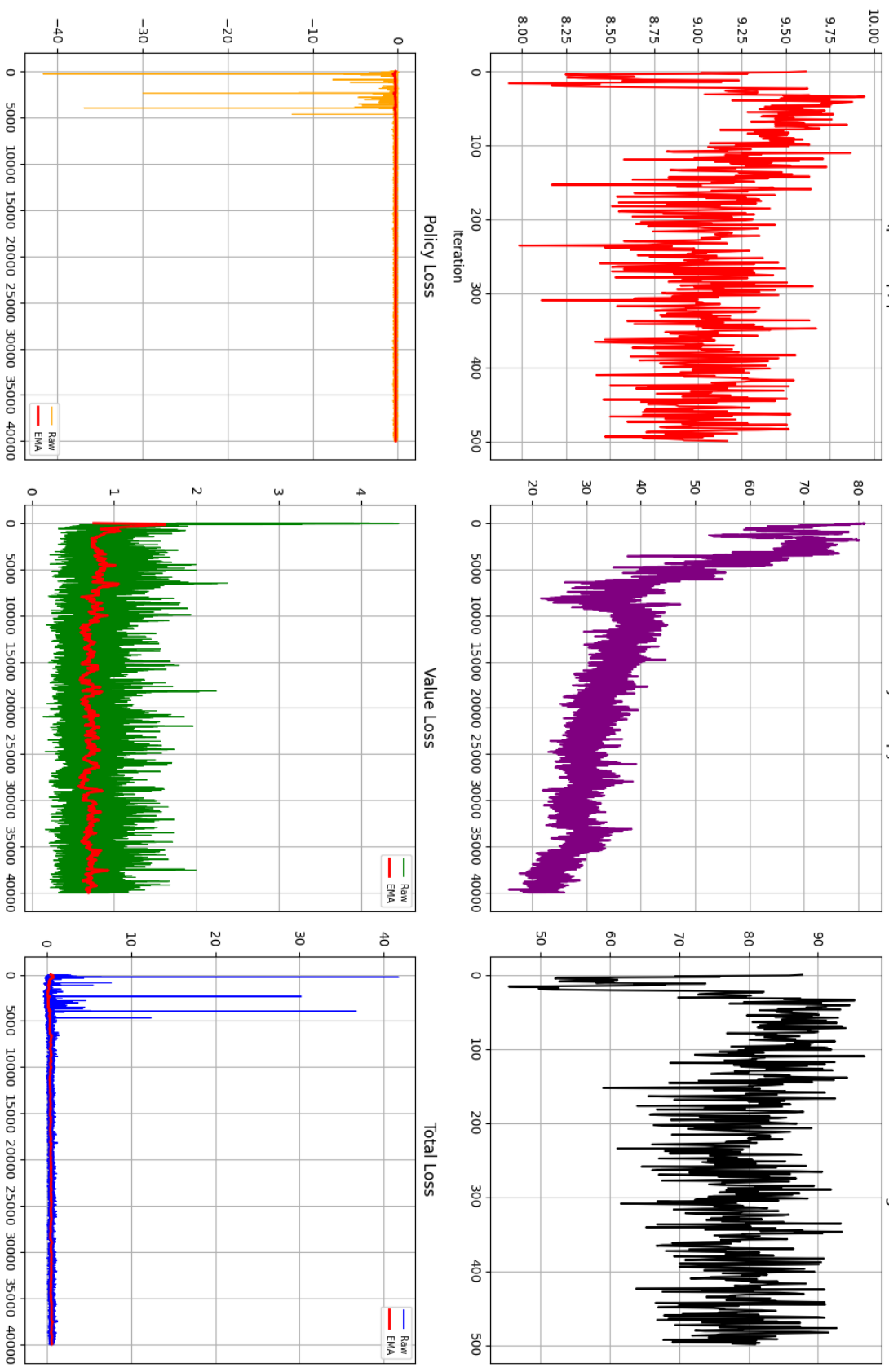


Figure A.1: Raw training metrics for the GAT architecture.

images/training/gath1.png


images/training/gcn1.png

images/training/gt1.png

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY