Evaluating Multi-Agent Modeller Representations

Jonathan Demke
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Physical Sciences and Mathematics Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/9727

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Evaluating Multi-Agent Modeller Representations

Jonathan Demke

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Master of Science

Dan Ventura, Chair
Jacob Crandall
Quinn Snell

Department of Computer Science
Brigham Young University

Copyright © 2022 Jonathan Demke
All Rights Reserved
ABSTRACT

Evaluating Multi-Agent Modeller Representations

Jonathan Demke
Department of Computer Science, BYU
Master of Science

The way a multi-agent modeller represents an agent not only affects its ability to reason about agents but also the interpretability of its representation space as well as its efficacy on future downstream tasks. We utilize and repurpose metrics from the field of representation learning to specifically analyze and compare multi-agent modellers that build real-valued vector representations of the agents they model. By generating two datasets and analyzing the representations of multiple LSTM- or transformer-based modellers with various embedding sizes, we demonstrate that representation metrics provide a more complete and nuanced picture of a modeller’s representation space than an analysis based only on performance. We also provide insights regarding LSTM- and transformer-based representations. Our proposed metrics are general enough to work on a wide variety of modellers and datasets.

Keywords: multi-agent modelling, representation learning, metrics
Thank you to my parents for their love, support, and prayers. Thank you to my advisor Dan Ventura for your support, patience, and for constantly helping me through my struggles and doubts. I would also like to thank Robert Morain for his willingness to always help out, for his great ideas which helped propel this thesis, and for being a good inspiration and friend.
Table of Contents

1 In Preparation: Evaluating Multi-Agent Modeller Representations 1
Chapter 1

In Preparation: Evaluating Multi-Agent Modeller Representations

This manuscript has not yet been accepted for publication.
Evaluating Multi-Agent Modeller Representations

Jonathan Demke\textsuperscript{1*}, Robert Morain\textsuperscript{1} and Dan Ventura\textsuperscript{1}

\textsuperscript{1}Computer Science Department, Brigham Young University.

*Corresponding author(s). E-mail(s): jpd0057@byu.edu;

Abstract

The way a multi-agent modeller represents an agent not only affects its ability to reason about agents but also the interpretability of its representation space as well as its efficacy on future downstream tasks. We utilize and repurpose metrics from the field of representation learning to specifically analyze and compare multi-agent modellers that build real-valued vector representations of the agents they model. By generating two datasets and analyzing the representations of multiple LSTM- or transformer-based modellers with various embedding sizes, we demonstrate that representation metrics provide a more complete and nuanced picture of a modeller’s representation space than an analysis based only on performance. We also provide insights regarding LSTM- and transformer-based representations. Our proposed metrics are general enough to work on a wide variety of modellers and datasets.

Keywords: Multi-agent modelling, representation learning, metrics

1 Introduction

The field of agent modelling has grown to include various modelling techniques as well as the ability to model various properties of agent behavior, such as future actions, beliefs, intents, etc. \cite{1}. Some recent work in the field of multi-agent modelling has analyzed the utility of multi-agent modeller representations in downstream tasks \cite{2–4}. Representation learning has shown the efficacy of good representations, because the way a model learns to represent its data can not only affect the performance of the model, but it can also improve the interpretability of the data \cite{5, 6} and be useful for downstream tasks or
multi-task problems [7]. Within the context of multi-agent systems (MAS), learning good representations for agents can be useful for making inferences on new agents, predicting agent behavior in new environments, and analyzing the interactions between groups of agents. Good representations can also help researchers reason about the space of possible agents, as well as the space of agents’ beliefs and intents, by being clear and interpretable. Because of the advantages of good representations, we propose some metrics that researchers can use to analyze their multi-agent modeller’s representation spaces. Our metrics require the representations to be real-valued vectors, so we focus on modellers that build real-valued representations of the agents they model.

While some work has focused on using representation learning as a source of inspiration for how to train an agent modeller [2, 4], we focus on representation learning metrics that have been used to analyze representations generally [5, 6, 8] and on promoting these metrics specifically for the field of multi-agent modelling. We discuss each metric in detail and describe a process researchers can use to compare two or more modellers’ representations.

We use these metrics to compare LSTM- [9] and transformer- [10] based modellers with various embedding sizes on two generated datasets and discuss the trends that the modellers exhibit on both datasets. We show that measuring the performance of a modeller is not a sufficient proxy for evaluating representation quality and conclude that medium-sized LSTMs give us consistently high scores on multiple metrics on both datasets. We also discuss future work that will help researchers analyze modeller representations further.

2 Performance vs. Representation Metrics

We analyze modeller capability through two lenses: performance metrics and representation metrics. Performance is typically some measurement of how well the multi-agent modeller reasons about modelled agents. These measurements typically evaluate and score the modeller’s predictions and are usually what the modeller is trained to improve on (e.g. action prediction loss). Representation metrics are measurements that look at the representation space a modeller creates. Instead of considering the modeller’s predictions directly, they evaluate the latent space (e.g. agent representation) that is formed while making a prediction. A good representation space is typically a byproduct of improving a modeller’s prediction ability, but some learning paradigms may also focus on optimizing the representation space directly. The reason we measure performance metrics when analyzing representations is twofold: first, the main focus of these multi-agent modellers is to reason about agents, so if they perform poorly, then the representations may be irrelevant; and second, a good representation positively influences the performance of a modeller [7], so performance metrics are indirect measures of representations.

---

1 We use the terms embedding and representation interchangeably.
2 Code: https://github.com/demkejon001/multi_agent_model_representation
2.1 Performance Metrics

Performance metrics depend on both dataset and modeller—one modeller may predict future rewards and measure performance with mean squared error, while another could predict next actions and measure performance with classification accuracy. When modellers share similar performance metrics, they can be directly compared. The most obvious example of this is comparing different hyperparameter settings for a single modeller; however, other possibilities include comparing different modeller types trained to optimize the same task or comparing modellers trained on different auxiliary tasks.

It’s more difficult to compare modellers with different performance criteria. One way to do this is to train the modellers on the same dataset to build their representation spaces and then evaluate their representations on the performance of a (downstream) no-shot, few-shot, or finetuning learning task [2–4]. In this work, we only address the former case, but we acknowledge the importance of future work focusing on the latter.

2.2 Representation Metrics

The representation metrics that we focus on are Disentanglement, Clustering, Stability, Convergence, and PCA Reduction. While some of these metrics can be applied directly to any dataset, others require information regarding the agent behaviors, intents, and desires within the dataset. This information is not used for training the modellers, but rather for evaluating their representation spaces by testing how well they disentangle and cluster the information. While these metrics may not be directly applicable for real-world datasets, we can get an idea of how a modeller will perform on real-world datasets, by evaluating them on generated datasets that provide the required information.

We define this information as an agent parameter space \( P \subseteq \prod_{\rho_1} \times \cdots \times \prod_{\rho_n} \), where a parameter \( \prod_{\rho_i} \) is uniquely identified by its label \( \rho_i \) and characterizes some factor of agent behavior. Then, a parameterization \( p \in P \) is an \( n \)-tuple \( (\rho_1 = \nu_1, \ldots, \rho_n = \nu_n) \) where \( \nu_1 \in \prod_{\rho_1}, \ldots, \nu_n \in \prod_{\rho_n} \). We can constrain the parameter space by fixing the value of one or more parameters; for example, \( \prod_{\rho_3 = \nu_3} \) is the subspace of \( P \) in which the third parameter always takes the value \( \nu_3 \). When the context is clear, we drop the parameter labels for simplicity, e.g., \( p = (\nu_1, \ldots, \nu_n) \). We use \( p_s \) to represent a specific agent using parameterization \( p \), since multiple agents can share a parameterization, but still act somewhat differently from each other. Here \( s \) represents any stochastic variance to which the agent may be subject, including the effects of hyperparameters, weight initialization, opponent behavior, or other environmental factors.

For the rest of the section, we discuss the motivation for the representation metrics as well as some relevant previous work; we give implementation details of all metrics in Section 6.
2.2.1 Disentanglement

Disentangling the underlying features of data has long been considered an important function of good representations [7]. Since the invention of VAE [11] and β-VAE [6], many systems for generating disentangled representations and metrics for analyzing them have been created. There are many definitions of disentanglement, but we have opted to use the definition from [5], which centers on two aspects: a representation is distributed, meaning independent factors from the dataset are represented by different dimensional subsets of the representation vector (a single dimension in a representation vector is called a code), and the encoded information is relevant and useful. For multi-agent modelling datasets, a factor would be an agent parameter $P_{\rho_i}$. The authors break the measure of disentanglement into three metrics:

1. Modularity: The extent to which a factor affects a subset of the representation space independently of other factors.
2. Compactness: The ability to represent a factor with a single code.
3. Explicitness: The ability to capture the relevant information of a factor.

2.2.2 Clustering

Clustering can be useful for elucidating a modeller’s agent embedding space by grouping together agents with similar behavior and potentially providing information about possible agent parameter spaces $P$. Since we are generating the datasets, we know $P$ a priori, so we would expect the embedding space to potentially create clusters based on the different agent parameters $P_{\rho_i}$ used to define $P$. We can get a notion of how well it clusters on agent parameters by partitioning the representation space based on agent parameterization and computing the silhouette score [12] on the partition.

Other work has also evaluated clustering using intra-inter cluster ratio (IICR) scores [2]. Quite similar to silhouette scoring, IICR scores are used to indicate whether trajectories from the same agent are closer together than trajectories from different agents. Our approach is similar to [2], but we use a more general approach by comparing the trajectories of agents that are similarly parameterized rather than the trajectories of a single agent.

We recognize that not all parameterizations in an agent parameter space will be accurately clustered in a modeller’s representation space; for example, two different parameterizations could have similar enough behavior that the modeller would place them closely in the embedding space, and capturing this inter-class similarity may be ideal behavior [13]. However, in general, we expect different parameterizations to exhibit higher inter-class variance and lower intra-class variance, meaning we don’t expect perfect silhouette scores, but we interpret higher scores as being generally correlated with meaningful clusterings.
2.2.3 Stability

Stability measures the effect of randomness on some outcome. Sources of randomness in agent modelling include modeller hyperparameter settings, weight initialization in modeller or agent, opponent behavior, and other stochastic environmental effects, for example. Typically stability is measured by the consistency of train/test performance, but we focus instead on consistent representations. How one computes the similarity between representations is an active research question. Representation similarity was initially measured with canonical correlation analysis, but more specialized measurement methods have been developed for comparing neural network representations [8, 14, 15]. One of these methods is centered kernel alignment (CKA) [8], which demonstrates the ability to make accurate assessments of representation similarity between networks trained with different initializations as well as different architectures. While some work has utilized these methods to understand how neural networks learn [15, 16] and how similar architecturally-different neural network representations can be [17], we demonstrate that CKA can also be used for the purpose of checking for consistent, stable representations.

2.2.4 Convergence

Because it is typically beneficial for a modeller to accurately reason about an agent as quickly as possible, the sooner a modeller can settle on an agent embedding, the better. We assume that a modeller forms a better agent embedding as it views more of the agent’s trajectories, so convergence is a measure of how similar intermediate agent embeddings are to the “final” agent embedding. Once again, we opt to use CKA to measure similarity among representations.

2.2.5 PCA Reduction

Dimensionality reduction is a useful tool for interpreting high-dimensional representation spaces. Other work has performed qualitative analyses on modeller representations by visualizing them after dimensionality reduction and showing that similar input data clusters together in the representation space [2, 4]. As with all our other metrics, we propose a quantitative metric using the popular dimensionality reduction algorithm, principal component analysis (PCA). PCA performs well on linear representations, meaning that if a modeller’s representation space is more linear than non-linear, and therefore simpler to interpret, it will be reduced easily with PCA. Our measurement looks at the number of dimensions needed to explain multiple levels of variance in a modeller representation.

3 Modelling an Agent

There are many approaches for building real-valued agent representations. Some modellers are trained alongside a reinforcement learner as an auxiliary task [18–20], some are trained on specifically multi-agent modelling tasks
[21, 22], others utilize semi-supervised techniques [2, 4], and often modellers are trained with multiple techniques.

The ToMnet [21] is an agent modeller that acts purely as an outside observer observing the environment. The ToMnet uses three neural networks, two of them are dedicated to building representations of the agent, and one is meant for predictive tasks. The ToMnet is shown to build a strong representation of an agent’s beliefs by testing it on the Sally-Anne [23] test for different agents.

To explore our metrics, we build two different datasets and describe their agent parameter spaces. Then we build multiple ToMnet modellers parameterized with two autoregressive networks: LSTMs and transformers, and three embedding sizes: 64 (small), 128 (medium), and 512 (large). We compare all modellers’ performance and representation metrics on both datasets.

4 Datasets

We perform all experiments on two datasets we generate from two types of Markov games. Generating such datasets allows us to control agent behavior and define an agent parameter space so that we can compare it to a modeller’s embedding space. A Markov game is represented as a 5-tuple \((N, S, A_1, T, R, \kappa)\) [24]. \(N\) is the set of agents, and in both types of Markov games used here \(|N| = 4\). \(S\) is the set of states. \(A_1\) is the vector of action sets, where each action set, \(A_i\), is the set of actions that agent \(i\) can take. \(T\) is a transition function that takes the current state and an action vector, which we interchangeably refer to as the joint action, to compute the next state, i.e. \(T : S \times A_1 \times A_2 \times \ldots \times A_{|N|} \rightarrow S\). The first agent, i.e. the agent with action set \(A_1\), is always the agent our modellers will reason about. We call this agent the *modelled agent* and all other agents *opponents* (even though they might be working cooperatively). \(R_{1:|N|}\) is the vector of reward functions, where each reward function \(R_i\) corresponds to agent \(i\). An agent’s reward function defines the real-valued output the agent receives during a state transition, i.e. \(R_i : S \times A_1 \times A_2 \times \ldots \times A_{|N|} \times S' \rightarrow \mathbb{R}\).

Each of our Markov games has a time horizon \(h\) that ends the game. For each game \(i\) that an agent \(p_s\) plays, we record an observation trajectory \(\tau_{1:h}^{p_s,i}\) containing observations for a modeller to see and a label trajectory \(y_{1:h}^{p_s,i}\) containing data for a modeller to predict. Depending on the dataset, the observation trajectory \(\tau_{1:h}^{p_s,i}\) is either a sequence of states, \([s_1^{p_s,i}, \ldots, s_h^{p_s,i}]\), or a sequence of state-joint-action pairs, \([((s_1^{p_s,i}, a_1^{p_s,i}), \ldots, (s_h^{p_s,i}, a_h^{p_s,i}))\], and the label trajectory \(y_{1:h}^{p_s,i}\) is either a sequence of actions the modelled agent took, \([a_1^{p_s,i}, \ldots, a_h^{p_s,i}]\), or a sequence of tuples containing the modelled agent’s actions and its goal consumption vectors \([(a_1^{p_s,i}, p_{1:1:12}^{p_s,i}), \ldots, (a_h^{p_s,i}, p_{h:1:12}^{p_s,i})]\). We discuss both trajectories further when we discuss our two datasets in detail.

A modelled agent will play ten games with the same opponents, and we store all ten observation trajectories in a list \(T_{1:10}^{p_s} = [\tau_{1:h}^{p_s,1}, \ldots, \tau_{1:h}^{p_s,10}]\) and all ten label trajectories in a list \(Y_{1:10}^{p_s} = [y_{1:h}^{p_s,1}, \ldots, y_{1:h}^{p_s,10}]\). Therefore, each
dataset is a set of tuples containing the lists of observation trajectories and label trajectories \((T_{1:n}^{p_s}, Y_{1:n}^{p_s})\) of each agent \(p_s\) we generate.

### 4.1 Iterative Action Dataset

The iterative action dataset is a simple multi-agent modelling task. It is similar to a 2-action matrix game; however, there are no payoffs so we can simplify the modelling task and focus only on a modeller’s ability to predict action patterns. The state space is \(S = \mathbb{B}^4 \cup \{-1, -1, -1, -1\}\). Each agent \(i\)'s action space is \(A_i = \{0, 1\}\), and all agents will select an action given the state. The game’s start state is \(s_1 = [-1, -1, -1, -1]\), and all subsequent states \(s_t\), for \(t > 1\), are the previous timestep’s action vector: \(s_t = a_{t-1:4}\). Since the game has no payoff, \(R_{i}\) is just the zero function. The game terminates after \(h = 20\) timesteps. An iterative action observation trajectory is the sequence of states from a game, i.e. \(\tau_{1:h}^{p_s,i} = [s_1^{p_s,i}, \ldots, s_h^{p_s,i}]\). We don’t need to store state-joint-action pairs in the observation trajectory because the states contain the agents’ actions. An iterative action label trajectory is the sequence of actions the modelled agent took, i.e. \(y_{1:h}^{p_s,i} = [a_1^{p_s,i}, \ldots, a_h^{p_s,i}]\).

#### 4.1.1 Iterative Action Agents

Each modelled agent is assigned an opponent to which it reacts in some way, and it reacts to the same opponent over all the games it plays. Each of the three opponents in the game is assigned a unique binomial distribution from which they randomly sample to determine which action they take, i.e. a mixed strategy, but the modelled agent’s behavior is determined by its parameterization \(p \in \mathbb{P}\).

The iterative action dataset’s agent parameter space is \(\mathbb{P} \subset \mathbb{P}_{\text{action}} \times \mathbb{P}_{\text{opp}} \times \mathbb{P}_{\text{trigger}} \times \mathbb{P}_{\text{pattern}} \times \mathbb{P}_{\text{strategy}}\). \(\mathbb{P}_{\text{action}} = \{0, 1\}\) is the starting action parameter, which is the action the modelled agent plays at the beginning of the game. \(\mathbb{P}_{\text{opp}} = \{2, 3, 4\}\) is the opponent index parameter, which represents the opponent (i.e. which action in the state) to which the modelled agent is reacting. \(\mathbb{P}_{\text{trigger}} = \{0, 1, \emptyset\}\) is the trigger action parameter. Not all agents are triggered by a trigger action, so a null parameter value \(\emptyset\) is included. \(\mathbb{P}_{\text{pattern}} = \mathbb{B}^3 \cup \mathbb{B}^4 \cup \{\emptyset\} - \{000, 0000, 111, 1111\}\) is the action pattern parameter. Note that \(\emptyset\) and \(\overline{\emptyset}\) are excluded because a modelled agent with starting action 0 and action pattern \(\emptyset\) or starting action 1 and action pattern \(\overline{\emptyset}\) would appear to be playing a pure strategy. Not all agents have an action pattern, so \(\emptyset\) is included. \(\mathbb{P}_{\text{strategy}}\) is the strategy parameter, which takes one of the following values:

1. Mirror: Plays its start action and then plays its opponent’s last action.
2. Grim Trigger: Plays its start action and when its opponent plays the trigger action the agent will play the opposite of its start action for the rest of the trajectory.
3. Trigger Pattern: Plays its start action and when its opponent plays the trigger action the agent will play its trigger pattern. Once it finishes playing its trigger pattern it resumes playing its start action until triggered again.
4. Win-Stay-Lose-Shift (WSLS): Plays its start action and every time its opponent plays a trigger action it will flip the current action it is playing.

Note that \( \mathbb{P} \neq \mathcal{P}_{\text{action}} \times \mathcal{P}_{\text{opp}} \times \mathcal{P}_{\text{trigger}} \times \mathcal{P}_{\text{pattern}} \times \mathcal{P}_{\text{strategy}} \), because we don’t allow invalid parameterizations in \( \mathbb{P} \), e.g. \((0, 1, 1, 010, \text{Mirror}) \notin \mathbb{P}\) because there are no mirror strategy agents that have an action pattern or trigger action. All modelled agents have non-null parameter values drawn from \( \mathcal{P}_{\text{action}}, \mathcal{P}_{\text{opp}}, \mathcal{P}_{\text{strategy}} \) but only some agents have non-null parameter values drawn from \( \mathcal{P}_{\text{trigger}} \) and \( \mathcal{P}_{\text{pattern}} \) because these are dependent on their strategy \( \mathcal{P}_{\text{strategy}} \). \( |\mathbb{P}| = 270 \) because there are \(|\mathcal{P}_{\text{action}}| \times |\mathcal{P}_{\text{opp}}| = 6 \) parameterizations with the mirror strategy, \(|\mathcal{P}_{\text{action}}| \times |\mathcal{P}_{\text{opp}}| \times |\mathcal{P}_{\text{trigger}} - \{\emptyset\}| = 12 \) parameterizations with the grim trigger strategy, \(|\mathcal{P}_{\text{action}}| \times |\mathcal{P}_{\text{opp}}| \times |\mathcal{P}_{\text{trigger}} - \{\emptyset\}| = 12 \) parameterizations with the WSLS strategy and \(|\mathcal{P}_{\text{action}}| \times |\mathcal{P}_{\text{opp}}| \times |\mathcal{P}_{\text{trigger}} - \{\emptyset\}| \times |\mathcal{P}_{\text{pattern}} - \{\emptyset\}| = 240 \) parameterizations with the trigger pattern strategy.

When we generate our dataset we randomly sample a parameterization based on strategy (e.g. \( p \sim \mathbb{P}_{\text{strategy}=\text{Mirror}} \)) and create a modelled agent \( p_s \) with the sampled \( p \). We sample 800 agents with the mirror strategy, 800 with the grim trigger strategy, 800 with the WSLS strategy, and 1600 with the trigger pattern strategy (because there are many more trigger pattern parameterizations), which gives us a dataset of 4000 \( T^{p_s}_{1:10} \).

### 4.2 Gridworld

The gridworld dataset is made up of randomly generated gridworlds, where the objective of the game is to consume goals. Each gridworld is a \( 21 \times 21 \) grid with walls on the border and \( 1 - 4 \) randomly generated internal walls, twelve random goal positions, and four random agent starting positions. A gridworld state space \( S \subseteq \mathbb{R}^{17 \times 21 \times 21} \) is used to represent 17 feature channels each of dimension \( 21 \times 21 \). Each feature channel uses the value 1 to indicate the presence of a feature and 0 to indicate its absence. The first channel contains the wall locations, the next twelve channels have each goal location, the following channel has the modelled agent’s location, and the last three channels have each opponent’s location. An agent \( i \)’s action space is \( A_i = \{\text{Up, Down, Left, Right, Stay}\} \). The transition function has deterministic dynamics. Any agent can consume a goal by entering its grid location, which removes the goal from the gridworld. Agents are unable to pass through walls or through other agents. The game ends at time horizon \( h = 20 \). Because of the random generation, certain worlds may have unreachable goals; however, each agent has at least one reachable goal. The agents’ reward functions \( R_i \) is a part of the agent parameter space \( \mathbb{P} \), similar to [21] and is described in the next section.
A gridworld observation trajectory is a sequence of state-joint-action pairs, i.e. $\tau_{ps,i}^{1:h} = [(s_{ps,i}^1, \vec{a}_{ps,i}^1), \cdots, (s_{ps,i}^h, \vec{a}_{ps,i}^h)]$, where $\vec{a}_{ps,i}^t \in \mathbb{B}^20$ is a Boolean representation of $a_{t,1:4}^{ps,i}$ obtained by concatenating each agent’s one-hot encoded action, i.e. $\vec{a}_{t,1:4}^{ps,i} = \bigoplus_{j=1}^4 \text{OneHot}(a_{t,j}^{ps,i})$. A gridworld label trajectory is the sequence of the modelled agent’s actions and goal consumption vectors, i.e. $y_{1:h}^{ps,i} = [(a_{1,1}^{ps,i}, \kappa_{1,1:12}^{ps,i}), \cdots, (a_{h,1}^{ps,i}, \kappa_{h,1:12}^{ps,i})]$. A goal consumption vector, $\kappa_{t,1:12}$, represents whether the modelled agent will consume a goal between timestep $t$ and $h$, so that the modeller learns to predict the future goals an agent will consume. Note that for a goal $g$ that is consumed at timestep $i$, all goal consumption values $\kappa_{t \leq i,g} = 1$ because the modelled agent will eventually consume $g$ at timestep $i$, and $\kappa_{t > i,g} = 0$ because the agent already consumed the goal, i.e. the agent won’t consume an already consumed goal.

4.2.1 Gridworld Agents

The agent parameter space is $\mathbb{P} = \mathcal{P}_{\text{reward}} \times \mathcal{P}_{\text{rank}} \times \mathcal{P}_{\text{collab}}$, where $\mathcal{P}_{\text{reward}}$ is the reward function parameter, $\mathcal{P}_{\text{rank}}$ the goal ranking function parameter, and $\mathcal{P}_{\text{collab}}$ the collaboration parameter. An agent parameterization determines the goals an agent pursues, and the agent will use A* pathfinding to navigate the gridworld to its goals. A modelled agent and its opponents will all share the same parameterization.

We limit $\mathcal{P}_{\text{reward}}$ to only six cooperative reward functions to simplify $\mathbb{P}$. The reward functions are cooperative reward functions, meaning that if any agent consumes a goal, then all agents will receive the same reward. Reward values are assigned by splitting the twelve goals into three fixed groups of four, with each group assigned one of the reward values from $\{1.0, 0.5, 0.1\}$. Each permutation of $\{1.0, 0.5, 0.1\}$ represents a different way of assigning reward values to the groups of goals. Since there are six permutations of $\{1.0, 0.5, 0.1\}$, there are six reward functions.

There are three goal ranking functions in $\mathcal{P}_{\text{rank}}$, each of which map a goal $g$’s reward value $r_g$ and distance from agent $d_g$ to a preference value $v_g$. If $g$ is unreachable or is already consumed then $v_g = -\infty$. Otherwise, the three goal ranking functions are defined as:

1. Highest: $v_g = r_g - (\epsilon \times d_g)$, where $\epsilon$ is a small value, so that goals with the highest reward are preferred, and if two goals have the same reward value, then the closer one is preferred.
2. Closest: $v_g = -d_g + (\epsilon \times r_g)$ so that the closest goals are preferred, and if two goals are the same distance away, then the one with the higher reward is preferred.
3. Discounted: $v_g = r_g \times \gamma^{d_g}$, where $\gamma$ is the discount value, so that the highest discounted rewards are preferred. We chose $\gamma = .75$ since it empirically had a good balance between close goals with low reward values and far goals with high reward values.

$\mathcal{P}_{\text{collab}} = \{\text{True, False}\}$, signifies whether the agent is collaborative or not. If a group of agents are collaborative, then they will combine their goal preferences
into a bipartite graph, where the agents and goals are the disjoint sets and their preferences are the weighted edges. The group of agents then use linear sum assignment on the bipartite graph to assign agents’ goals based on the group’s goal preferences. For example, if two collaborative agents, using a closest goal ranking function, share a highest preference for the same goal, then whichever agent has a higher secondary goal preference will go after the secondary goal and leave the highest preferred goal to the other agent. If the group is not collaborative, then an agent will always select its most preferred goal, independent of other agents’ preferences, e.g. in the example above, both agents would attempt to consume the same goal.

There are 36 agent parameterizations since $|\mathcal{P}| = |\mathcal{P}_{\text{reward}}| \times |\mathcal{P}_{\text{rank}}| \times |\mathcal{P}_{\text{collab}}|$. When we generate our dataset, we randomly sample a parameterization based on collaboration (e.g. $p \sim \mathcal{P}_{\text{collab} = \text{True}}$) to generate a modelled agent and its opponents. We sample 1024 collaborative agents and 1024 non-collaborative agents, which gives us a dataset of 2048 $T_{1:10}$.

5 The ToMnet Modeller

The ToMnet [21] is an agent modeller that acts purely as an outside observer observing the environment. The ToMnet consists of three neural networks: a character network, a mental network, and a prediction network. The character network is used to reason about the past trajectories of an agent and form an embedding that captures the characteristics of the agent, called the character embedding, $e_{\text{char}}$. The mental network looks at the current trajectory of an agent as well as $e_{\text{char}}$ to form a mental embedding, $e_{\text{mental}}$, meant to represent the current mental state of the agent. The prediction network considers both $e_{\text{char}}$ and $e_{\text{mental}}$ as well as the modelled agent’s current state, to make various predictions, such as the next action the agent will take and the future goals it will consume. Our ToMnet implementation, which has slight variations from the original, is described below and is visualized in Fig. 1.
To acquire the necessary inputs—$T_{\text{past}}$, $\tau_{\text{cur}}$, and $s_c$—for training, we randomly sample an agent’s list of trajectories $T_{1:103}$ from our dataset. To do so, we first randomly sample the number of past trajectories, $n \sim [0, \ldots, 5]$, and the current timestep to be predicted, $c \sim [1, \ldots, h]$. We can then randomly sample a list, $I_{1:n+1}$, of $n + 1$ indices from $[1, \ldots, 10]$ without replacement. This allows us to randomly sample $n + 1$ observation trajectories $\tilde{T}_{1:n+1} = [T_{i_1}, \ldots, T_{i_{n+1}}]$ and label trajectories $\tilde{Y}_{1:n+1} = [Y_{i_1}, \ldots, Y_{i_{n+1}}]$ and ensure that $\tilde{Y}_i$ is $\tilde{T}_i$’s corresponding label trajectory. The modeller will treat $\tilde{T}_{1:n} = [\tau_{1h}, \ldots, \tau_{nh}]$ as the past observed trajectories, $\tau_{n:c-1}^{n+1}$ as the current trajectory, $s_c^{n+1}$ as the current state, and predict $y_c^{n+1}$.

Each modeller includes functions $\phi_o$ and $\phi_s$ which map an observation $\tau_t^i$ and the current state $s_c^{n+1}$, respectively, to real-valued vector representations. Since each dataset’s observations and states are different, $\phi_o$ and $\phi_s$ are dependent on the dataset, so we describe them in Sections 5.1 and 5.2. A pre-processing function $F_\tau$ converts a full or partial trajectory into a sequence of observation representations using $\phi_o$: $F_\tau(\tau_1^i) = [\phi_o(\tau_1^i), \ldots, \phi_o(\tau_t^i)]$.

The sample $\tilde{T}_{1:n}$ is converted into $T_{\text{past}}$, the input to our character network, by preprocessing each trajectory using $F_\tau$ and concatenating them into a single sequence with a learned embedding $\xi$, of length $|\phi_o(\tau_1^i)|$, marking the border between trajectories: $T_{\text{past}} = [\xi] \oplus F_\tau(\tau_1^h) \oplus [\xi] \oplus F_\tau(\tau_1^2) \oplus \cdots \oplus F_\tau(\tau_1^h) \oplus [\xi]$. The current trajectory is then $\tau_{\text{cur}} = [\xi] \oplus F_\tau(\tau_{1:c-1}^{n+1})$ and the current state is $s_c = \phi_s(s_c^{n+1})$. If $n = 0$, then $T_{\text{past}} = [\xi]$ and if $c = 1$, then $\tau_{\text{cur}} = [\xi]$.

We pass $T_{\text{past}}$ into our character network, sequentially processing one element at a time. The character network comprises of a fully connected layer and an autoregressive network. The fully connected layer linearly resizes an element in $T_{\text{past}}$ to be of length $|e_{\text{char}}|$ and passes it to the autoregressive network. The autoregressive network’s final output becomes the agent’s character embedding $e_{\text{char}}$.

The mental network, like the character network, contains a fully connected layer and an autoregressive network. Each element $\tau_i$ in $\tau_{\text{cur}}$ is passed sequentially into the mental network. However, before this is done $e_{\text{char}}$ is concatenated to it: $\tau_i \oplus e_{\text{char}}$. This concatenation is then passed into the mental network’s fully connected layer, which linearly resizes it to be of length $|e_{\text{mental}}|$ and passes it to the autoregressive network. The autoregressive network’s final output becomes the agent’s mental embedding $e_{\text{mental}}$.

The prediction network is a shared torso network with different sub-prediction network heads. There is a sub-prediction network for every labelled data type in $y_c^{n+1}$, which means the iterative action dataset only requires an action prediction network head, while the gridworld dataset has both action prediction and goal prediction network heads. The prediction network’s shared torso takes $s_c \oplus e_{\text{mental}} \oplus e_{\text{char}}$ as input and uses a fully connected network to map the input into a new representation so that each sub-prediction network head can use it to make their respective predictions.

---

To improve readability, when the context is clear, we will forgo using $p_s$ and let it be implied.
An action prediction network head takes the prediction torso network’s output and uses a fully connected network with a softmax at the end to predict the modelled agent’s action distribution, \( \hat{\pi}^{n+1}_c \). The next step action prediction loss is defined as the negative log-likelihood of the true action, given by \( a^{n+1}_{c,1} \in y^{n+1}_c \), taken with respect to \( \hat{\pi}^{n+1}_c \):

\[
L_{\text{action}} = -\log \hat{\pi}^{n+1}_c (a^{n+1}_{c,1} \mid s_c, e_{\text{mental}}, e_{\text{char}})
\]

The goal consumption prediction network head takes the prediction torso network’s output and uses a fully connected network with a sigmoid activation to predict the modelled agent’s goal consumption vector \( \hat{\kappa}^{n+1}_{c,1:12} \). \( \hat{\kappa}^{n+1}_{c,g} \) therefore represents the predicted likelihood that the modelled agent will consume goal \( g \) at the current timestep \( c \) or after it. The goal consumption prediction loss is the negative log-likelihood of the consumption of goal \( g \), given by \( \kappa^{n+1}_{c,g} \in y^{n+1}_c \), with respect to the predicted goal consumption distribution, \( \hat{\kappa}^{n+1}_{c,g} \):

\[
L_{\text{consumption}} = \sum_g -\log \hat{\kappa}^{n+1}_{c,g} (\kappa^{n+1}_{c,g} \mid s_c, e_{\text{mental}}, e_{\text{char}}) \ast (4\kappa^{n+1}_{c,g} + 1)
\]

where \((4\kappa^{n+1}_{c,g} + 1)\) is a weight to improve goal consumption recall. We focus on recall since most goals in a gridworld will not be consumed by an agent.\(^4\)

To improve the training gradient and efficiency, the ToMnet modeller can always be trained on \( n = 5 \) past trajectories by using the character network’s autoregressive output at each location of \( \xi \) in \( T_{\text{past}} \). Each \( e_{\text{char}} \) outputted by the character network then represents a character embedding built with increasing numbers (0 to 5) of past trajectories. Similarly, the ToMnet can also be trained on the entire current trajectory, rather than (only) at a specific timestep \( c \). For further training details refer to Appendix A. We now describe the ToMnet modellers we used to train on each dataset.

### 5.1 Iterative Action ToMnet

Both \( \phi_o : \mathbb{Z}^4 \to \mathbb{Z}^4 \) and \( \phi_s : \mathbb{Z}^4 \to \mathbb{Z}^4 \) are identity functions because the iterative action state space is so small that the character and mental network’s fully connected layers are sufficiently able to generate latent representations for observations and states.

#### 5.1.1 Modeller parameterization

We create six different parameterizations of iterative action ToMnets. Three use the LSTM architecture as the autoregressive network with character embedding sizes 64, 128, and 512, and a fixed mental embedding size 64. The

\(^4\)There are twelve goals split between four agents and not all goals are accessible to an agent, therefore most of the values in \( \kappa^{n+1}_{c,g} \) will be zero.
Table 1: Modeller parameterizations used on the iterative action dataset

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Character Network</th>
<th>Mental Network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Embed Size</td>
<td>Layers</td>
</tr>
<tr>
<td><strong>LSTM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>Transformer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>512</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Modeller parameterizations used on the gridworld dataset

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Character Network</th>
<th>Mental Network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Embed Size</td>
<td>Layers</td>
</tr>
<tr>
<td><strong>LSTM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>Transformer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

other three use the transformer architecture as the autoregressive network with the same character and mental embedding sizes as the LSTM architecture; refer to Table 1. For each of the six iterative action ToMnet parameterizations, we initialize and train five different modellers with different seeds, creating a total of 30 different modellers.

### 5.2 Gridworld ToMnet

Because observations are made up of high-dimensional state-joint-action pairs, we use a convolutional network to convert the states into a real-valued vector, \( \phi_s: \mathbb{B}^{17 \times 21 \times 21} \rightarrow \mathbb{R}^{64} \), and a fully connected layer to convert the joint actions into a real-valued vector, \( \phi_a: \mathbb{B}^{20} \rightarrow \mathbb{R}^{8} \). \( \phi_o: \mathbb{B}^{17 \times 21 \times 21} \times \mathbb{B}^{20} \rightarrow \mathbb{R}^{72} \) is then the concatenation of the latent representations for a given state-joint-action pair, i.e. \( \phi_o(s^i_t, a^i_t) = \phi_s(s^i_t) \oplus \phi_a(a^i_t) \).

#### 5.2.1 Modeller parameterization

Like the iterative action ToMnets, we initialize and train five modellers for each gridworld ToMnet parameterization, refer to Table 2, creating a total of 30 modellers.

---

5 We use Huggingface’s GPT2 configuration [25] to parameterize our transformer modeller, except we don’t use the word token embedding network that comes with GPT2, since \( \phi_o \) will produce the embeddings, but we do use the default positional encoder.
6 Metric Implementations

In order to compute the metrics for modeller $m$, we need to generate a test dataset. First, we generate a list, $P_{1:d}$, of $d$ agents to be modelled, containing $k$ agents for every $p \in \mathbb{P}$, i.e. $d = k \times |\mathbb{P}|$. Each of the $k$ agents that share a parameterization $p$ will have different seeds to generate either different opponent mixed strategies (for the iterative action test dataset) or different gridworlds (for the gridworld dataset). For every modelled agent $p_s \in P_{1:d}$ we generate opponents, as described in Sections 4.1 and 4.2, and let the agents play a series of $n$ games. We record $\tau_{p_s,i}$ and $y_{p_s,i}$ for each of the $n$ games and store them in $T_{p_s}$ and $Y_{p_s}$ respectively. We use $n = 5$. A modeller $m$ can then perform two functions on $T_{p_s}$: predicting agent behavior, $\rho^m$, and converting an agent’s trajectories into an agent embedding, $\epsilon^m$, which are utilized for the performance and representation metrics respectively.

6.1 Performance

We use $\rho^m(T_{p_s}^n)$ to retrieve $\hat{Y}_{p_s,m}^n$, which in the case of our ToMnet modellers means that trajectories $T_{p_s}^{n-1}$ will be used for $T_{past}$, and $T_{p_s}^n$ will be used for $\tau_{cur}$ and the current state. Instead of sampling $c \in [1, \cdots, h]$ though, we use every $c \in [1, \cdots, h]$ to generate multiple $\tau_{cur}$ and $s_c$ from $T_{p_s}^n$ and average our results over all time steps. For every $c \in [1, \cdots, h]$ we can get the ToMnet’s predicted action distribution $\hat{\pi}_{c,p_s,m}^n$ and if applicable the predicted goal consumption prediction $\hat{\kappa}_{c,p_s,m}^n$. We can then calculate the modeller’s action loss ($L_{action}$) and accuracy (using maximum likelihood), and if applicable the goal consumption loss ($L_{consumption}$) and recall (also using maximum likelihood) at each timestep $c$ and average the results over all $h$ timesteps. Each metric can then be aggregated across each agent in $P_{1:d}$, e.g.:

\[
\frac{1}{h \times d} \sum_{p_s \in P_{1:d}} \sum_{c=1}^{h} L_{action}(T_{past}, \tau_{cur}, s_c)
\]

6.2 Representation

We use a modeller’s $\epsilon^m$ function to retrieve the embedding of each modelled agent in $p_s \in P_{1:d}$, which in the case of ToMnet modellers would entail preprocessing $T_{1:n}$ into $T_{past}$ and passing it into the character network to get $\epsilon_{char}$, which we will now represent as $\epsilon_{p_s}^m$ (the representation of agent $p_s$ given by modeller $m$). Then we record $E^m$ which is a matrix of agent embeddings, where the row denotes the modelled agent in $P_{1:d}$ and the column denotes the number of trajectories that were used to create the embedding:

---

$^6$Because $|\mathbb{P}|$ is small and contains only discrete-valued parameters for the experiments described here, it is possible to exhaustively cover the entire parameter space. For larger $|\mathbb{P}|$, some form of sampling should be applied.

$^7$On datasets where modelled agents do not play separate games but rather a single continuous game, the game can be partitioned into $n$ chunks of $h$ timesteps, i.e. $\tau_{1:h} = \tau_{h(i-1):h}$.
\[ E^m = \begin{bmatrix} e^m([T^1_1]) & \cdots & e^m(T^1_{1:n}) \\ \vdots & \ddots & \vdots \\ e^m([T^d_1]) & \cdots & e^m(T^d_{1:n}) \end{bmatrix} = \begin{bmatrix} e^m_{P_1,1} & \cdots & e^m_{P_1,n} \\ \vdots & \ddots & \vdots \\ e^m_{P_d,1} & \cdots & e^m_{P_d,n} \end{bmatrix} \]

In what follows, \( E^m_{p,i} \) is the entire column \( i \) in \( E^m \). We index the rows of \( E^m \) using \( p_s \) and \( p \) as keys which map to indices such that \( E^m_{p,i} \) represents an element of \( E^m \), where \( E^m_{p,1:i} \) is equivalent to \( E^m_{j,i} \) when \( p = \tilde{P}_j \). \( E^m_p \) is the selection of all rows in \( P_{1:d} \) that use parameterization \( p \) and \( E^m_{p,i} \) is the selection of all elements from column \( i \) that use parameterization \( p \).

### 6.2.1 Disentanglement

We adhere to the recommendation of [5] and measure modularity, compactness, and explicitness with the DCI Random Forest algorithm\(^8\) [26]. It’s important to note that disentanglement metrics are based on independent agent parameters in the dataset which can, in general, be difficult to determine [5]. It is simple for our datasets, however, because they are synthetic, with any parameter dependence explicitly designed. For the iterative action dataset, independent parameters are opponent index, starting action, and agent strategy (trigger action and action pattern depend on agent strategy); for the gridworld dataset, all agent parameters are independent.

At a high level the DCI Random Forest algorithm trains \(|p|\) random forest regressors, each one predicting a specific factor\(^9\) of \( \mathbb{P} \) given any \( E^m_{p,i,n} \in E^m \). Let \( f = |p| \) represent the number of factors in \( \mathbb{P} \) and \( c = |E^m_{p,i,n}| \) represent the number of codes in a representation, i.e. the embedding size. Using the “feature importance” calculated from the random forests, DCI Random Forest generates a collection of probabilities, \( P_{1:c,1:f} \), where \( P_{1:c,j} \) is a probability distribution and \( P_{i,j} \) describes the relative importance of code \( i \) for predicting factor \( j \) compared to all other codes. It also generates \( \tilde{P}_{1:c,1:f} \), where \( \tilde{P}_{i,1:f} \) is a probability distribution describing the relative importance of code \( i \) for predicting factor \( j \) (\( \tilde{P}_{i,j} \)) compared to all other factors. The modularity of a code \( i \) is then \( M_i = 1 - H_f(P_{i,:}) \), where \( H_f(P_{i,:}) = -\sum_{j=1}^{f} P_{i,j} \log_{f} P_{i,j} \). The overall modularity score is a weighted average over \( M_i \), where weight is determined by the relative importance of code \( i \) for predicting anything in all of the random forests. The compactness of factor \( j \) is \( C_j = 1 - H_c(\tilde{P}_{:,j}) \), where \( H_c(\tilde{P}_{:,j}) = -\sum_{i=1}^{c} \tilde{P}_{i,j} \log_{c} \tilde{P}_{i,j} \). The overall compactness score is the average over all \( C_j \). The explicitness of factor \( j \) is \( I_j = \max(1 - 12 \times \text{mse}_j, 0) \) where \( \text{mse}_j \) is the mean squared training error of the random forest trained on factor \( j \). The overall explicitness score is the average over all \( I_j \). For more details refer to [5].

---

\(^8\)DCI stands for Disentanglement, Completeness, and Informativeness and is an alternative naming convention for Modularity, Compactness, Explicitness respectively.

\(^9\)Because all of our factors are categorical and won’t work with regression, we convert each factor in \( p \) to be a one-hot vector.
6.2.2 Clustering

We can get a notion of how well a modeller \( m \)’s representation clusters according to agent parameters by partitioning the representation space based on agent parameterization and computing the silhouette score on the partition:

\[
\text{Silhouette}(Q) = \frac{1}{d} \sum_{E_{m,n} \in Q} \sum_{E_{m,n}' \in E_{m,n}} \frac{b(E_{m,n}') - a(E_{m,n}')}{\max(a(E_{m,n}), b(E_{m,n}'))}
\]

where \( Q \) is a partition of \( E_{m,n} \), \( d \) is the length of \( E_{m,n} \), \( a() \) computes the average intra-cluster distance between point \( E_{m,n} \) and cluster \( E_{p,n} \), and \( b() \) computes the average nearest-cluster distance between point \( E_{m,n} \) and the nearest cluster \( E_{q,n} \), \( q \in Q, q \neq p \).

When agent parameters fail to be differentiated in the any of the modellers’ representation spaces, silhouette scores are negatively affected and no objective comparison between modellers can be performed. To automatically filter these unused parameters, we search for a more informative parameter space, over a set of parameter spaces, across the set of all modellers \( M \).

The set of parameter spaces \( \mathcal{P} \) is built by taking the Cartesian product of each set in the powerset of agent parameters, excluding the empty set. For example, using the gridworld parameter space, \( \mathcal{P} = \mathcal{P}_{\text{rank}} \times \mathcal{P}_{\text{reward}} \times \mathcal{P}_{\text{collab}} \), \( \mathcal{P} = \{ \mathcal{P}_{\text{reward}} \times \mathcal{P}_{\text{rank}} \times \mathcal{P}_{\text{collab}}, \mathcal{P}_{\text{reward}} \times \mathcal{P}_{\text{rank}}, \mathcal{P}_{\text{reward}} \times \mathcal{P}_{\text{collab}}, \mathcal{P}_{\text{rank}} \times \mathcal{P}_{\text{collab}}, \mathcal{P}_{\text{reward}}, \mathcal{P}_{\text{rank}}, \mathcal{P}_{\text{collab}} \} \). We can then find a more informative parameter space with \( \arg\max \sum_{m \in M} \text{Silhouette}(\{ E_{m,n} | p \in \mathcal{P}' \}) \) and report the silhouette scores using that parameter space.\(^{10}\)

6.2.3 Stability

Stability is measured by computing the average CKA similarity between the set of representations generated by modellers in the set \( M_1 \) and the set of representations generated by modellers in the set \( M_2 \):

\[
\text{Stability}(M_1, M_2) = \frac{\sum_{E_{i,n} \in M_1} \sum_{E_{i,n}' \in M_2} \frac{\text{CKA}(E_{i,n}, E_{i,n}')}{|M_1| \times |M_2| - |M_1 \cap M_2|}}{|M_1| \times |M_2| - |M_1 \cap M_2|}
\]  

(1)

Since we are computing an average, the stability metric will return a value from 0 to 1 indicating the similarity between modellers’ representation spaces. When \( M_1 = M_2 \), stability is being measured relative to all stochastic variation that affects the set of models \( M_1 \); when \( M_1 \neq M_2 \), stability is being measured relative to any stochastic variation that differentiates \( M_1 \) and \( M_2 \).

6.2.4 Convergence

We measure how quickly a model \( m \) converges to its final representation using CKA to compare each intermediate representation with the final one:

\( ^{10}\)Here, we abuse notation and take \( p \in \mathcal{P}' \) to mean we are ignoring any parameters not represented in the Cartesian product (e.g., if \( \mathcal{P}' = \mathcal{P}_{\text{rank}} \times \mathcal{P}_{\text{reward}} \), then the parameter \( \mathcal{P}_{\text{collab}} \) is ignored). This may, of course, result in multiple \( p \in \mathcal{P} \) being clustered together in \( \mathcal{P}' \).
Convergence($E^m$) = \[ n^{-1} \sum_{i=1}^{n-1} \frac{\text{CKA}(E^m_{:,i}, E^m_{:,n})}{n} \] (2)

A larger average difference across these comparisons indicates faster convergence.

### 6.2.5 PCA Reduction

Recall that every entry in $E^m_{:,n}$ is an embedding vector so $E^m_{:,n}$ is a 2D matrix of real values to which PCA can be applied. We use PCA to measure how many dimensions are needed to explain 50%, 75%, 80%, and 90% of the variance in $E^m_{:,n}$, giving an idea of the efficiency of the embedding as well as possibly some sense of the linearity of the representation manifold.

### 7 Results

We applied our metrics on 30 trained gridworld ToMnets and 30 trained iterative action ToMnets. All the modellers were trained until there was no improvement in overall validation loss. Sections 7.1 and 7.2 describe the generated test datasets used to compute our metrics and the later subsections summarize the results for each metric described in Section 6. We plot each modeller’s results on a strip plot and use mean lines to aggregate similarly parameterized modellers for all metrics except stability, which is visualized as a heatmap.

#### 7.1 Iterative Action

The iterative action test dataset is made up of $k = 100$ agents per agent parameterization $p$. As explained in Section 4.1.1, $|P| = 270$ with most of the parameterizations using the trigger pattern strategy. Because using all $p \in P$ would skew the results toward the prediction and representation of trigger pattern strategy agents, we limit the trigger pattern strategy agents in our test dataset by generating only $k$ agents that had the action pattern 001, $k$ agents that had the action pattern 0011, and $k$ agents that had any action pattern in $P_{\text{pattern}}$ other than 001 or 0011. Thus the number of parameterizations, in our test dataset, utilizing the trigger pattern strategy is $|P_{\text{action}}| \times |P_{\text{opp}}| \times |P_{\text{trigger}}| \times 3 = 36$. Along with the six mirror strategy parameterizations, twelve grim trigger strategy parameterizations, and twelve WSLS strategy parameterizations, we have a total of 66 parameterizations in our test dataset, so $d = k \times 66 = 6600$.

#### 7.2 Gridworld

The gridworld test dataset is made up of $k = 300$ agents per agent parameterization $p$. As explained in Section 4.2.1, $|P| = 36$, therefore $d = 36 \times k = 10800$. 

18
Fig. 2 Iterative Action Performance Metrics: All modellers share roughly similar performance, except for the large transformer modeller.

7.3 Performance

Most of the modellers have similar performance on the iterative action dataset (refer to Fig. 2). An exception is the large transformer, which performs slightly worse than the other modellers. The gridworld ToMnets’ performance tells a slightly different story in Fig. 3. Most of the modellers’ performance scores are still similar, but the small and medium transformers perform slightly better than the LSTMs. Once again, the large transformer suffers in performance compared to all the other modellers and even more significantly than it did on the iterative action dataset. It may be possible that a better training protocol (e.g., larger batch sizes) could improve its performance.

7.4 Disentanglement

Fig. 4 shows that LSTMs generally exhibit better results than transformers over the disentanglement metrics. Both LSTMs and transformers have similar levels of explicitness, but the LSTM is more modular and compact overall. On the iterative action dataset, an increase in LSTM embedding size is beneficial on all disentanglement metrics, but the gridworld dataset shows that large LSTMs have decreased modularity. Note that even though the large transformer’s performance metrics on both datasets were worse than all other modellers, its disentanglement metrics, on the iterative action dataset, are either as good as or better than the small and medium transformer modellers.

In contrast, on the gridworld dataset, the large transformer has the worst disentanglement scores, especially in the case of explicitness—which is close to zero, meaning that the character embedding isn’t capturing any information about the gridworld agent parameter space. One of the reasons that all modellers have poorer explicitness scores on the gridworld dataset than on the iterative action dataset is because the random forests were unable to predict the gridworld reward function parameter, meaning that none of the modeller’s character embeddings had any information regarding agent reward functions. It is unclear whether the reward functions are unnecessary for modeller’s to make predictions or whether this is a weakness in the ToMnet itself.
Gridworld Performance Metrics: As with the iterative action data set, the large transformer also has the worst performance on the gridworld dataset; however, in the gridworld scenario, the small- and medium-sized transformers performed slightly better than their LSTM counterparts.

7.5 Clustering

For the iterative action ToMnets, the original agent embedding space $P \subset P_{\text{action}} \times P_{\text{opp}} \times P_{\text{trigger}} \times P_{\text{pattern}} \times P_{\text{strategy}}$ proved to be more informative than any of the other potential parameter spaces. The resulting silhouette scores for the clusters are shown in Fig. 5. For the gridworld ToMnets, the embedding space $P = P_{\text{rank}} \times P_{\text{collab}}$ was more informative than any of the potential parameter spaces. This is supported by the gridworld explicitness scores, which were zero for the agent reward function parameter. Therefore, the gridworld silhouette scores shown in Fig. 5, are based only on clusters that partition the gridworld test dataset by goal ranker type and whether the agent is collaborative or not.

All modellers show similar silhouette scores on the iterative action dataset, but on the gridworld dataset the transformer modellers have degrading scores the larger they get. In fact, the large transformer silhouette score on the gridworld dataset is less than zero, meaning there is no separation among clusters, which corroborates its explicitness score in Fig. 4. Despite the iterative action large transformer modellers’ performance metrics being slightly worse than the other modellers, it has similar silhouette scores; and despite the LSTMs’ performing slightly worse than the transformers on the gridworld
Most the modellers share similar explicitness scores, but LSTMs typically exhibit better modularity and compactness. The large transformer does well on the iterative action dataset but performs poorly on the gridworld dataset. Given the transformers degrading silhouette scores on the gridworld, it seems that LSTMs cluster slightly better, according to the \textit{a priori} parameterizations, than transformers; and no LSTM embedding size is obviously better than another.

### 7.6 Stability

We computed the stability of differently initialized modellers by putting the representations of all modellers that share a modeller parameterization, as described in sections 5.1 and 5.2, into a set and plugging that set as both arguments in Equation 1 (i.e. as both $M_1$ and $M_2$). To measure the effect of embedding size on stability, pairwise stability is computed between two sets that have the same autoregressive architecture, but different embedding sizes. Applying pairwise stability is a useful tool for when the effects of model
Fig. 5 Clustering - Iterative Action (left), Gridworld (right): Silhouette scores for clustering the iterative action embedding spaces based on all possible agent parameters: agent strategy, starting action, opponent index, trigger action, and action pattern. Silhouette scores for clustering the gridworld embedding spaces based on: goal ranker and whether or not the agent was collaborative. All modeller scores are similar on the iterative action dataset, but on the gridworld dataset the transformer results degrade as the embedding size increases.

The stability scores are visualized as heatmaps in Fig. 6. The diagonal results show the effects of weight initialization on stability, which we will call self-stability for brevity; and the off-diagonal results show the pairwise effects of embedding size on stability, which we will call size-stability. Other than the large transformer, most modellers have similar self-stability and size-stability, with the large LSTM and medium transformer having the best self-stability. The large transformers though, have relatively poor stability scores. However, notice that the large transformer self-stability is higher than its pairwise size-stability scores on the iterative action dataset, but lower on the gridworld dataset. This might indicate that the large iterative action transformers potentially build different representations of the agents than their smaller transformer counterparts. The large transformer representations may be noisy, but they are more similar to each other than they are with the smaller transformer counterparts. Unfortunately, the large gridworld representations are so noisy that we are unsure whether it is making similar or different representations compared to its smaller counterparts.

Fig. 7 visualizes a cross-architecture comparison of representation spaces. The diagonal results can be thought of as the effects of autoregressive architecture on stability. Other than the large transformer, the representation spaces of the different gridworld architectures are very similar, and they are more similar than the iterative action cross-architecture comparisons. The large iterative action transformer representations however, are more similar to its LSTM counterparts than the large gridworld transformer is to its LSTM counterparts, most likely due to the noisy representations of the large gridworld transformer. A possible explanation for why gridworld representations are more similar to each other could be because there are fewer agent parameterizations in the gridworld dataset and therefore fewer possible representations for the data.
Fig. 6 Stability - Iterative Action (left), Gridworld (right): All modellers have similar stability scores, except for the large transformer, which has dissimilar representations from its smaller transformer counterparts and even from other large transformers (with different weight initializations).

Fig. 7 Cross architecture comparison - Iterative Action (left), Gridworld (right): Comparing LSTM representation spaces with transformer representation spaces. Gridworld comparisons show that the modellers had very similar representations, potentially because of the simplicity of the gridworld parameter space; by the same reasoning, the fact that the iterative action parameter space is more complex may explain why the iterative action comparisons are not as similar.
7.7 Convergence

Excluding the large transformer, the LSTM and transformer modellers’ convergence values, shown in Fig. 8, are quite similar. LSTMs perform slightly better on the iterative action dataset and slightly worse on the gridworld dataset. The large transformer converges well on the iterative action dataset but not on the gridworld dataset, mirroring its disentanglement results. Again it should be noted that although the large transformer’s performance metrics were the worst on the iterative action dataset, it has better disentanglement metrics and faster convergence than its smaller transformer counterparts.

7.8 PCA Reduction

In both datasets the number of dimensions needed to explain the PCA variability in an LSTM increases with embedding size; in fact, the correlation between LSTM embedding size and the number of dimensions needed to explain 80% of PCA variability is \( \approx 0.97 \) on the iterative action dataset and \( \approx 0.88 \) on the gridworld dataset. Fig. 9 shows the number of dimensions needed to explain 80% of the PCA variability. The results are comparable to 50%, 75%, and 90% which can be viewed in the Appendix (Fig. A1).

7.9 Representation as a Distribution

Visualizing a modeller’s representation space as a distribution can help reveal the effect that embedding size and autoregressive architecture have on an embedding space. The representation space is visualized as a distribution by making a boxplot for each code in \( E_{m,n} \) and ordering the boxplots by their median values. Visualizations for iterative action ToMnet representations can be seen in Fig. 10 and for gridworld ToMnet representations in Fig. 11.

Both datasets’ LSTM modellers have very sparse representations since most of their dimensions are zero-centered and exhibit little variance. It’s possible that the sparsity in LSTMs means that only specific codes are being used to
represent specific agent parameters—the number of non-zero dimensions does not appreciably increase when the embedding size is doubled, perhaps implying a local representation. This is supported by the fact that the gridworld ToMnet representations, which model an agent parameter space smaller than the iterative action ToMnets, are far sparser than the iterative action ToMnet representations. Furthermore, [27] has shown that a single LSTM code can be responsible for a specific dataset factor.

The transformer representations, like the LSTM, are mostly zero-centered, but they exhibit more variance across all dimensions. In fact, the iterative action representations have very consistent variance on all dimensions and that variance goes down with an increase in embedding size. The Spearman rank correlation for transformer embedding size and the variance of a code is $-0.74$ and $-0.64$ on the iterative action and gridworld representations respectively. We chose Spearman rank correlation over Pearson correlation due to the outliers found in the large transformer representations. An example of a few codes with extremely high variance can be seen in a large transformer’s representation in Fig. 10.

It’s possible that while LSTMs may have local representations, transformers have distributed representations that spread information about the agent across all of its dimensions. If that is the case, then large transformer modellers have less variance than their smaller counterparts because they are distributing the agent information across more parameters and similarly, gridworld transformer representations have less variance than their iterative action counterparts because they are representing a smaller agent parameter space. Given this theory, we can infer that large transformer modellers have poorer representation metrics on the gridworld dataset because too few agent parameterizations have to be distributed across the large transformer’s representation space, creating a very sparse representation, which may be more sensitive to noise.
Fig. 10 Iterative Action Representations as a Distribution - We visualize a modeller’s representation space as a boxplot for each dimension in $E_{m : n}$. LSTM representations exhibit sparse representations, while the transformer representations have consistent variance on all dimensions. The ratio of sparse to non-sparse decreases with increased LSTM size denote a local representation. The transformer representations maintain mostly consistent variance across all dimensions, but its variance decreases as with increased embedding size.

7.10 Recommending a Modeller

Based on only performance metrics, small- or medium-sized transformer modellers appear to be the best choice for these tasks since they had similar performance to LSTMs on the iterative action dataset and slightly better performance on the gridworld dataset. However, when also considering the representation metrics, LSTMs may be the better choice—the transformers
Gridworld Representations as a Distribution - We visualize a modeller’s representation space as a boxplot for each dimension in $E_{i,n}^m$. The magnitude of the variance for both gridworld representation types is markedly lower than that of the iterative action representations, perhaps due to modelling a smaller agent parameter space.

didn’t disentangle their representations as well as the LSTMs; and the silhouette scores for the gridworld dataset show degrading performance with increased transformer size. The one metric where transformer modellers vastly outperformed their LSTM counterparts was PCA reduction, however, upon retrospection, PCA reduction may not provide much useful information. It tells us that transformer modellers give us easily reducible linear representations and the LSTM modellers do not, but given the sparsity of the LSTMs embeddings, it’s possible that most of the dimensions are tiny random values that share no correlation and therefore the PCA dimensionality will increase...
with embedding size even though there are just a few dimensions which are being used by the LSTM to capture all the pertinent information. So while the LSTM representations could be reduced to their few non-sparse dimensions, PCA reduction cannot convey that.

The medium-sized LSTM ToMnet may be the best overall modeller; all LSTM modellers have roughly similar results on all metrics, but the medium-sized LSTM ToMnet performed best on multiple metrics across both datasets. While the large-sized LSTM performed slightly better on more metrics than the medium-sized LSTM, its relatively low gridworld modularity score may make it a potentially riskier choice. Our recommendation for transformer modellers would be to keep the embedding size relatively small since increasing the embedding size of an adequately performing modeller may cause adverse effects on its representation space. Of course, this analysis is based on two relatively simple datasets, and testing on additional and more complex datasets may change the outcomes, but the purpose of this paper is to show how utilizing representation metrics allows for more informed decision-making regarding multi-agent modeller selection.

8 Conclusion

In this work we have described metrics that can be used on multi-agent modeller representations to analyze their performance, their ability to disentangle and cluster agent behavior, and their stability, convergence, and reducibility. We describe a process of generating a dataset, based on an agent parameter space $\mathbb{P}$, to analyze a representation and fairly compare multiple representations. We demonstrate that representation metrics provide a more complete and nuanced picture of a modeller’s representation space, than would an analysis based only on performance metrics, by applying them on multiple ToMnet modellers across two generated datasets. The metrics allow us to not only pick a suitable modeller but also help diagnose issues with modellers’ representations. Our approach is general enough to work on a wide variety of modellers and datasets.

Directions for future work include further generalization of the proposed metrics for different modellers and datasets as well as investigating the utility of additional metrics. As mentioned earlier, research on pertinent one-shot, few-shot, or finetuning downstream tasks to further measure the utility of representations will be important. Furthermore, the PCA reduction metric was useful for seeing that transformers give easily reducible linear representations, but its scope is potentially too limited since the sparse nature of the LSTM representations should admit a natural reduction in dimensionality; further investigation of how different modeller representations interact with different reduction techniques may shed additional light on how well a representation can be reduced. Certain metric implementations also make the assumption that agents have a fixed behavior and that their parameterization $p$ is static,
but agent behavior may evolve over time, requiring metrics like convergence to account for dynamic parameterizations.

Different levels of generalization could also be incorporated when analyzing representations. For example, our work both trains and tests a modeller on agents that are derived from the entire agent parameter space, and even though there are sources of stochasticity which require generalization from the training set to the test set, a modeller trained this way will experience all the agent behaviors on which it will be tested. As a next step, it would be interesting to explore representation generalization by training a modeller on one subset of $\mathbb{P}$ and testing on another subset, further assessing a modeller’s ability to generalize. In addition, it may be possible to analyze different levels of agent-interaction as described in [28].

While we discuss looking at variations of modeller size and architecture, other avenues of research may consider variations of training protocol that can help improve representation spaces, such as learning rate or batch sizes. Since high performance doesn’t guarantee the best representations, training modellers past the point of no loss improvement could be beneficial for learning simpler and better representations, so long as it doesn’t lead to overfitting. It could also be advantageous to analyze auxiliary modelling tasks that can improve a modeller’s representation.

References


To improve the ToMnet’s gradient and sample efficiency during training, we make use of all $\xi$ embeddings in $T_{\text{past}}$ and the full sequence of observations and predictions in $T_{\text{cur}}$. This means we don’t sample $n$ or $c$, but rather set them to 5 and $h$ respectively. We preprocess our observations as normal to get $T_{\text{past}}$ and $\tau_{\text{cur}}$, but the current state is set up differently.

We use $e_{\text{char},i}$ to represent an agent’s character embedding formed by using $i-1$ trajectories, which means the first $\xi$ embedding in $T_{\text{past}}$ will give us $e_{\text{char},1}$, since it is at the very beginning of $T_{\text{past}}$ and has no trajectory information. The second $\xi$ embedding gives us $e_{\text{char},2}$ and so forth, up to $e_{\text{char},6}$. We then make a matrix $E_{\text{char},1:6,1:h}$, which copies $e_{\text{char},i}$ $h$ times along the column dimension, one for every timestep in the current trajectory:

$$E_{\text{char},1:6,1:h} = \begin{bmatrix}
e_{\text{char},1} & \cdots & e_{\text{char},1} \\
e_{\text{char},2} & \cdots & e_{\text{char},2} \\
e_{\text{char},3} & \cdots & e_{\text{char},3} \\
e_{\text{char},4} & \cdots & e_{\text{char},4} \\
e_{\text{char},5} & \cdots & e_{\text{char},5} \\
e_{\text{char},6} & \cdots & e_{\text{char},6}
\end{bmatrix}$$

Since we are dealing with six unique $e_{\text{char},i}$ we need to copy $\tau_{\text{cur}}$ six times. Allow $T_{\text{cur},1:6,1:h}$ to represent the six copies of $\tau_{\text{cur}}$ where each row is a copy, but the columns represent the timesteps in $\tau_{\text{cur}}$:

$$T_{\text{cur},1:6,1:h} = \begin{bmatrix}
\xi \phi_o(\tau_{6}^{0}) & \cdots & \phi_o(\tau_{6}^{h-1}) \\
\xi \phi_o(\tau_{6}^{1}) & \cdots & \phi_o(\tau_{6}^{h-1}) \\
\xi \phi_o(\tau_{6}^{2}) & \cdots & \phi_o(\tau_{6}^{h-1}) \\
\xi \phi_o(\tau_{6}^{3}) & \cdots & \phi_o(\tau_{6}^{h-1}) \\
\xi \phi_o(\tau_{6}^{4}) & \cdots & \phi_o(\tau_{6}^{h-1}) \\
\xi \phi_o(\tau_{6}^{5}) & \cdots & \phi_o(\tau_{6}^{h-1})
\end{bmatrix}$$

The input to the mental network can then be $T_{\text{cur},1:6,1:h} \oplus E_{\text{char},1:6,1:h}$, by concatenating $T_{\text{cur},i,j}$ with $E_{\text{char},i,j}$. The mental network will then output a
mental embedding for every element in $T_{\text{cur},1:6,1:h}$ resulting in matrix $E_{\text{mental},1:6,1:h}$, where $E_{\text{mental},i,j}$ represents a mental embedding built using $e_{\text{char},i}$ at timestep $j - 1$ in $\tau_{1:h}^6$.

We set up the prediction network’s input state differently because we use all the states in $\tau_{1:h}^6$, and since we are dealing with six unique $e_{\text{char},i}$ we make six copies of the states in $\tau_{1:h}^6$:

$$S_{1,1:6h} = \begin{bmatrix}
\phi_s(s_1^6) \\
\vdots \\
\phi_s(s_h^6) \\
\vdots \\
\phi_s(s_1^6) \\
\vdots \\
\phi_s(s_h^6)
\end{bmatrix}$$

and we reshape $E_{\text{char},1:6,1:h}$ and $E_{\text{mental},1:6,1:h}$ to be $1 \times 6h$ matrices:

$$E_{\text{mental},1:1:6h} = \begin{bmatrix}
e_{\text{mental},1,1} \\
\vdots \\
e_{\text{mental},1,h} \\
\vdots \\
e_{\text{mental},6,1} \\
\vdots \\
e_{\text{mental},6,h}
\end{bmatrix} \quad E_{\text{char},1:1:6h} = \begin{bmatrix}
e_{\text{char},1} \\
\vdots \\
e_{\text{char},6}
\end{bmatrix}$$

The prediction network then receives $S_{1,1:6h} \oplus E_{\text{mental},1:1:6h} \oplus E_{\text{char},1:1:6h}$ as input. Because the learned embedding $\xi$ is the first timestep of $\tau_{\text{cur}}$, $E_{\text{mental},1,i}$ is one timestep behind $S_{1,i}$ and therefore no information is leaked to the prediction network regarding future states or actions.
Fig. A1  Number of dimensions needed to explain 50%, 75%, and 90% of variability with PCA - Iterative Action (left), Gridworld (right)