D1.2 Simple Metacognitive Model

Tool innovation using discrete state space active inference

Poppy Collis, Paul F Kinghorn and Christopher L Buckley

MT-018 v 0.0 — 2022/12/08

Contents

1 Introduction

This codebase provides a minimal model of tool innovation that involves an active inference agent making generalised inferences about the tool structure required to solve an extension-of-reach task. We provide detailed documentation on the various parameters within the code that you can adjust and experiment with as well as specifying the setup for the following three experiments run in [Collis et al., 2023]:

1. Tool use: we show that with a perfect generative model, the agent can straightforwardly use tools optimally to solve a task (see Section. 3.3.1).
2. Tool discovery: we demonstrate that the agent can discover the correct tools and learn to solve the task when it is not provided with this information in its model *a priori* (see Section. 3.3.2).

3. Tool innovation: we show that factorising the hidden states of the generative model into the affordances of the tool can enable the agent to conceive offline the appropriate properties of the tool required to solve the task (see Section. 3.3.3).

For a comprehensive understanding of the methodologies and theories associated with this codebase, please refer to our research paper [Collis et al., 2023]. The paper provides in-depth insights and discussions on the concepts, experiments, and findings that form the basis of this project. We encourage readers and users of this repository to consult the paper for a more detailed exposition of the work presented here.

2 Fundamentals

This codebase uses the Python package *pymdp* for modelling active inference agents in discrete state space [Heins et al., 2022]. For more detailed documentation on the package itself, we refer the reader to the official documentation [1].

2.1 Required packages

Table. 1 lists the required Python libraries to be installed for the code to run.

Table 1: Required Python Libraries

Package
numpy
matplotlib
scipy
pandas

2.2 Environment

The task environment for the extension-of-reach task is shown in Fig. 1. It consists of a 2 x 4 grid of locations in which the agent can only move between two rooms: left and right (shown here in grey). The agent is always initialised in the left-hand room. A vertical tool (V) is located in the left-hand room while a horizontal tool (H) is located in the right-hand room. In one of the remaining rooms (shown in blue), a reward is located, and it is the goal of the agent to try and reach this reward using the tools provided.

For example, if the reward is in the room directly north of the right-hand room as shown, the agent is required to be in the right-hand room holding tool V in order to reach it. The agent can choose to pick up the tool if it is in the relevant room, while it may drop tools whilst it is in any room (in which case, any of the tools in the agent's possession are dropped and returned to their original rooms). If the agent already possesses a tool and picks up a different tool, this creates a compound tool (HV). The rooms directly north, east and west of the left and right rooms are known as the adjacent rooms and these only require the individual tools V or H to solve. The northeastern and northwestern rooms are termed the corner rooms and present a greater challenge for the agent as they require the construction of the compound tool (HV) to solve.

The generative process is handled by the `tool_making_environment.py` script and the `toolEnv` class within it. This implements the 2-D grid-world shown in Fig. 1 and ensures that the agent receives appropriate observations while interacting with the world. In the codebase, the rooms are labelled with indices according to the mapping shown in Table. 2.

[1] Official *pymdp* documentation can be found here: `https://pymdp-rtd.readthedocs.io/en/latest/`

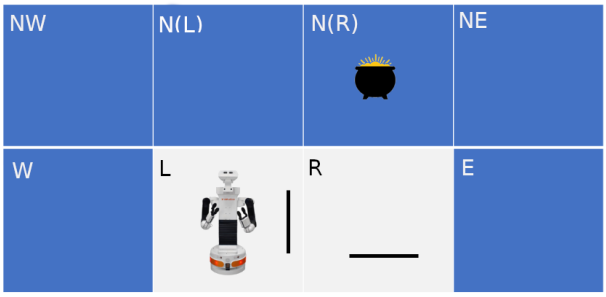


Figure 1: The task of the the agent is to reach the reward by using the tools provided. The simulation environment shows the agent (*robot*) can only move between the left and right rooms (*grey*) and the reward (shown as a pot of gold) can be placed in any of the other rooms (*blue*). A vertical tool (V) can be picked up in the left room, and a horizontal tool (H) in the right room

Table 2: Room indexing

Direction	Index
L	0
R	1
E	2
NE	3
NR	4
NL	5
NW	6
W	7

2.3 Constructing the generative model

We can construct an agent with one of two different generative model structures (the Tool State Model in `tool_state_model.py` and the Affordance Model in `affordance_model.py`). The generative model structures differ in terms of the way in which the observation of a tool is represented in the hidden states (i.e. how the agent models the latent causes of the observations) (see Fig. 2).

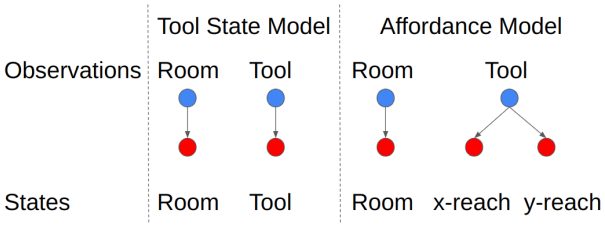


Figure 2: In the Tool State Model used of experiment 2, there is a one to one mapping between the tools the agent observes, and the internal representations it has for them. In the Affordance Model in experiment 3, the agent separates the latent tool states into properties of x-reach and y-reach

The agent's probabilistic generative model of the world (represented as a standard POMDP) consists of the following components.

1. Observations, o: the data points or evidence collected from the real world that the model aims to

Table 3: Generative Model Structures

	Observations	Hidden States	Control States
Tool State Model	obs_room	state_room	room_action
	obs_tool	state_tool	tool_action
	obs_reward		
Affordance Model	obs_room	state_room	room_action
	obs_tool	state_x_reach	x_action
	obs_reward	state_y_reach	y_action

explain.

2. Hidden States, s: these represent the agent's model of the latent common causes of observations obtained by a process of inference, which are not necessarily veridical representations of the causal structure of the world.

3. Control States, u: this is the behavioural output of the agent. The possible actions the agent can take are [`Null, Move, Pick-up, Drop`].

4. Policies, π: a fixed sequence of control states u_τ for each time-step τ that together represent a plan of action of length T, $\pi = \{u_1, ..., u_T\}$. For the simulations, we have a policy length of 4 time steps (`policy_len = 4`). All possible 4 step policies are restricted such that the agent can only enact one control state at a time (i.e. the variable `combined_action_list` is restricted to a one hot vector per time-step of a policy).

5. Model Parameters, ϕ: the entries of the categorical likelihood distributions.

The most important distributions when specifying this generative model are the observation likelihood $P(o_\tau|s_\tau;\phi)$, the transition likelihood $P(s_\tau|s_{\tau-1},\pi;\phi)$, and the prior preference over observations $P(o_\tau)$, known in *pymdp* as the A, B and C matrices respectively. The dimensions of these matrices are as follows:

- A Matrix: $observation_i \times state_0 \times state_1 \times state_2 \times ...$
- B Matrix: $state_i \times state_i \times state_j \times control_0$ (dependent on `B_factor_list` which specifies which states depend on which other states)
- C Matrix: $observation_i$

See Table. 4 for further detail on the shapes of these structures.

Table 4: Shapes of generative model data structures

	A Matrix	B Matrix	C Matrix	B Factor List
Tool State Model	A[0]=4×4×2	B[0]=4×4×2×4	C[0]=4	[[0,1],[1]]
	A[1]=2×4×2	B[1]=2×2×4	C[1]=2	
	A[2]=2×4×2		C[2]=2	
Affordance Model	A[0]=4×4×2	B[0]=4×4×2×4	C[0]=4	[[0,1],[1]]
	A[1]=2×4×2	B[1]=2×2×4	C[1]=2	
	A[2]=2×4×2		C[2]=2	

2.4 Factorisation

We factorise our representations of observations o_τ and states s_τ into separate modalities and factors: $o_\tau = \{o_\tau^1, o_\tau^2, \ldots, o_\tau^M\}$ and $s_\tau = \{s_\tau^1, s_\tau^2, \ldots, s_\tau^F\}$ in which M is the number of modalities and F the number of hidden state factors such that the likelihood distributions can be written as:

$$P(\mathbf{o}_\tau \mid \mathbf{s}_\tau) = \prod_{m=1}^{M} P(o_\tau^m \mid \mathbf{s}_\tau) \tag{1}$$

$$P(\mathbf{s}_\tau \mid \mathbf{s}_{\tau-1}, \mathbf{u}_{\tau-1}) = \prod_{f=1}^{F} P(s_\tau^f \mid \mathbf{s}_{\tau-1}, \mathbf{u}_{\tau-1}) \tag{2}$$

We allow state factors in the transition likelihoods to depend on themselves and a specified subset of other state factors (defined in the `B_factor_list`).[2]

2.5 Learning

Learning in active inference is a process of inference over the model parameters, ϕ, which are simply the categorical likelihood distributions. We treat these parameters as something over which the agent maintains and updates beliefs (i.e. as random variables). Consider the example of an A matrix, which encodes the observation likelihood model $P(o|s)$, with the entry $A[i, j]$ representing the probability of seeing observation i given state j. There is therefore a separate categorical distribution for each state (i.e. each column sums to 1). The Dirichlet distribution is a conjugate prior for the categorical distribution, and we therefore model prior beliefs over the categorical as a Dirichlet. It can be shown that, when the agent obtains new empirical information, the Bayesian process of updating this prior is simply a count-based increase of the Dirichlet parameters according to the observation o and inferred state s [Heins et al., 2022]:

$$\alpha_{posterior} = \alpha_{prior} + o \otimes s \tag{3}$$

where α represents the Dirichlet parameters.

The priors over the categorical likelihood distributions are Dirichlet distributions defined in the code as `pA` and `pB`. The count-based increase of the Dirichlet parameters is dependent on the `dir_scale` parameter which controls the learning rate.

We can test our agents in a **continual learning paradigm** in which the agent can be exposed to a number of different reward locations on given experimental trials. This point will prove crucial in our ability to show that our model is capable of a non-trivial notion of innovation (see Sections 3.3.3 and 3.3.2)

3 Usage

3.1 Variables

The following section provides a description of important variable in the codebase.

prob_A — The assigned probability to the 'true' state in the observation likelihood matrix

prob_B — The assigned probability to the 'true' state in the transition model matrix

fully_known — A Boolean variable indicating whether the matrix is constructed as a degenerate distribution with a priori knowledge of state transitions (True) or a uniform distribution (False).

[2] This requires a recent branch of *pymdp* which enables this kind of factorisation. See `https://github.com/infer-actively/pymdp/tree/sparse_likelihoods_111`

dir_scale — The scale of the counts used to update the Dirichlet concentration parameters in response to new evidence, thereby controlling the rate of learning (when they are high, new information leads to more substantial updates).

punish — Prior preference for the observation of punish in the reward modality in terms of relative log probabilities.

reward —Prior preference for the observation of reward in the reward modality in terms of relative log probabilities

policy_len — The number of future time-steps considered for each policy.

steps_per_run — The number of steps the agent can take during a single experimental run.

locations — A vector specifying the location of the reward for every run.

3.2 Algorithm

Listing 1: Pseudocode for active inference agent

```

A  <- create_A()
B  <- create_B()
pB <- create_pB()
C  <- create_C()
policies <- create_policies()
agent = Agent(A, B, pB, C, policies)

for i in range(runs):
    set_reward_location()
    agent.A = A # as new reward location will change A matrix each time
    env.reset()
    initial_observations()

    for j in range(steps):
        infer_states(observation)
        infer_policies()
        action <- sample_action()
        agent.B <- update_B()
        observation <- env.step(action)
```

3.3 Experiments

3.3.1 Tool use

In the first set of experiments, we provide the agent with a perfect probabilistic generative model of the world. This means that the correct transition likelihood and observation likelihood distributions are provided and therefore no learning is required.

Listing 2: Parameters for experiment 1

```
tool_state_model.py

fully_known = True # agent provided with perfect transition model
prob_B = 1.0
locations = [2,3,4,5,6,7] # try any combination of reward locations
```

3.3.2 Tool discovery

Whilst we again provide the agent with a fully known observation likelihood distribution, for the following experiment we initialise the agent with a uniformly distributed transition likelihood model. This means that the agent initially knows nothing about how states and actions effect future states. It therefore must learn these state transitions rather than being provided with this information from the outset (as in tool use experiment). In a continual learning task, we present the agent with the adjacent rooms (6,7,4,3) which only require the learning about H and V. Then test it on rooms 2 and 5 (which requires tool HV). We find that despite having knowledge about tools V and H, it must still take exploratory actions to 'discover' the composite tool HV.

Listing 3: Parameters for experiment 2

```
tool_state_model.py

fully_known = False # agent must learn transition model
# first expose agent to adjacent rooms before testing the corner rooms (2 or 5)
locations = [7,7,7,4,4,4,3,3,3,6,6,6,2]
```

3.3.3 Tool innovation

For the minimal model of tool innovation, we have the agent learn the entries of the transition likelihood distribution model from scratch (i.e. we initialise it as a uniform distribution as in experiment 2). However, our transition likelihood now includes the new factorised tool states (see Fig. 2). In a continual learning task, we present the agent with the adjacent rooms (6,7,4,3) which only require the learning about H and V. Then test it on rooms 2 and 5 (which requires tool HV). Despite never having seen rooms 2 or 5 before, the agent can leverage its current knowledge in order to compose relevant affordances and spontaneously 'invent' the new tool, solving the task optimally without need for 'discovering' it.

Listing 4: Parameters for experiment 3

```
affordance_model.py

fully_known = False # agent must learn transition model
# first expose agent to adjacent rooms before testing the corner rooms (2 or 5)
locations = [7,7,7,4,4,4,3,3,3,6,6,6,2]
```

For a comprehensive report of the results of the three experiments described above, please refer to our research paper [Collis et al., 2023].

3.4 Plots

The plotting functions are contained in the `plots.py` script.

plot_barchart() — Plots the expected free energy (EFE) components of utility (orange), parameter information gain (light green) and state information gain (dark green) for each policy at a particular timestep. The total EFE for each policy is represented by the black line and the selected policy is circled (see Fig. 3).

plot_tool_certainty() — Plots a measure of how well the agent knows each tool by looking at the posterior probability associated with the correct control state (i.e. action) for creating each tool when solving for rooms (see Fig. 4).

plot_policy_rankings() — Plots how the selected policy ranks in the context of all possible policies in terms of utility and information gain (see Fig. 5).

plot_num_steps_history() — Plots the number of steps taken to solve the task for each reward location (see Fig. 6).

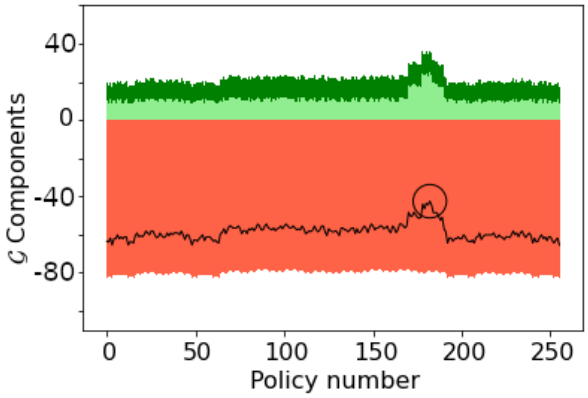


Figure 3: plot_barchart(): Decomposing expected free energy G into respective information gain and utility contributions can elucidate the agent's intended consequences of an action. The expected free energy G (black line) is evaluated over a set of 256 policies. The components which contribute to the selection of the best policy (circled) are state information gain (dark green), parameter information gain (light green) and utility (orange). Examples shown are instances when the selected policy is a) driven by information gain as there is little variation in utility and b) driven by utility as there is little variation in information gain

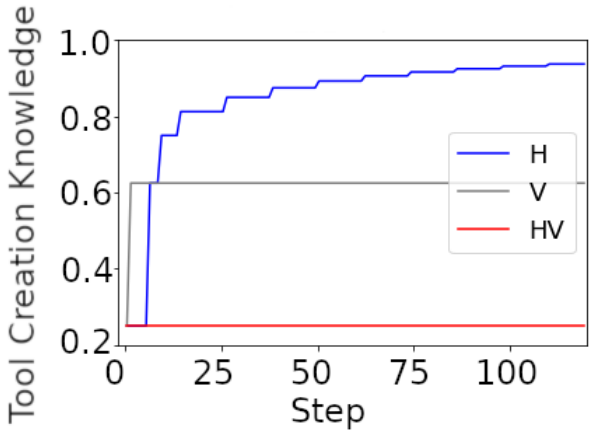


Figure 4: plot_tool_certainty(): The agent only learns the tools that it needs to learn in order to solve the task. We provide a measure of how well the agent knows each tool by looking at the posterior probability associated with the correct control state (i.e. action) for creating each tool when solving for rooms

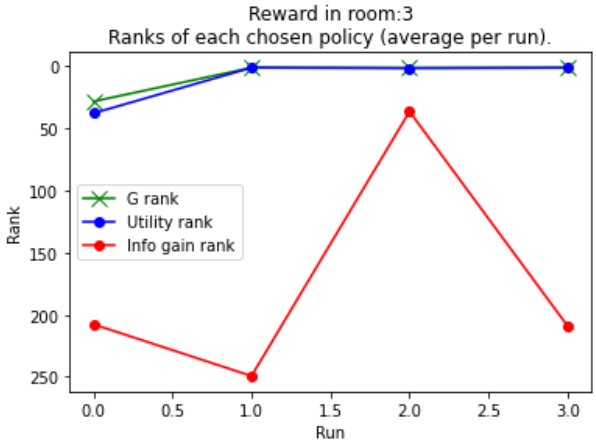


Figure 5: plot_policy_rankings(): Policy selection is initially dominated by information gain, but is then very quickly driven by utility as the agent learns new information. Graph shows how the selected policy ranks in the context of all possible policies in terms of utility and information gain (best rank is 0, worst is 256).

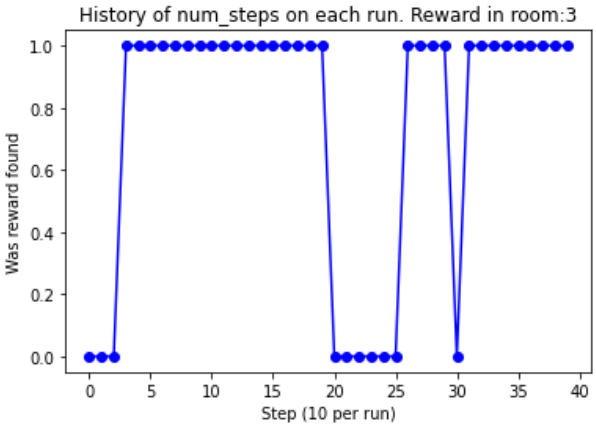


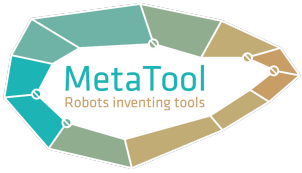
Figure 6: plot_num_steps_history(): The number of steps taken to solve the task for each reward location decreases quickly over runs to the optimal number of steps, reflecting the agent learning via discovery. Graphs show the number of steps to solve reward location

References

[Collis et al., 2023] Collis, P., Kinghorn, P. F., and Buckley, C. L. (2023). Understanding tool discovery and tool innovation using active inference.

[Heins et al., 2022] Heins, C., Millidge, B., Demekas, D., Klein, B., Friston, K., Couzin, I., and Tschantz, A. (2022). pymdp: A python library for active inference in discrete state spaces. *arXiv preprint arXiv:2201.03904*.

metatool-project.eu



Title: D1.2 Simple Metacognitive Model
Author: Poppy Collis, Paul F Kinghorn and Christopher L Buckley
Date: 2022/12/08
Reference: MT-018 v 0.0 Draft
URL:

European Innovation Council

The **MetaTool Project** is a research project funded by the EC Horizon Europe programme though grant HE #101070940. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the Horizon Europe programme. Neither the European Union nor the granting authority can be held responsible for them.