A Full Correlations for Analysis

In this section, we show full plots for the analysis in §5.3.1. We show

1. Average L2 norm of all module-states (trained and random weights) (Figure 10).
2. Comparison of average pair-wise correlation of module-states for trained and random weights (Figure 11).
3. Average pair-wise correlation between L2 norm of all module-states (Figure 12).
4. More in-depth plots of activity and attention coefficients (Figure 13).

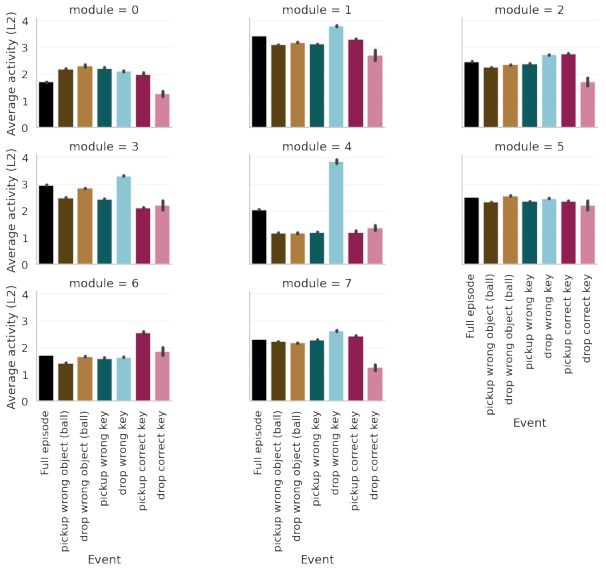

Figure 8: Trained weights.

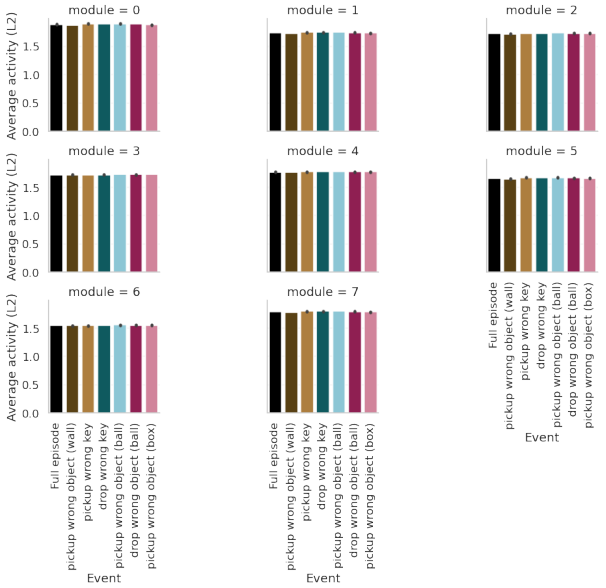

Figure 9: Random weights.

Figure 10: Average L2 norm of module-states. We find that when weights are trained on the task, some modules are selective for different events. For example, Module 4 is selective for "drop wrong key" and module 6 is selective for "pickup correct key". When we use random weights, we see that all modules have the same activity for all events. This indicates that they have not learned any task-specific activity.

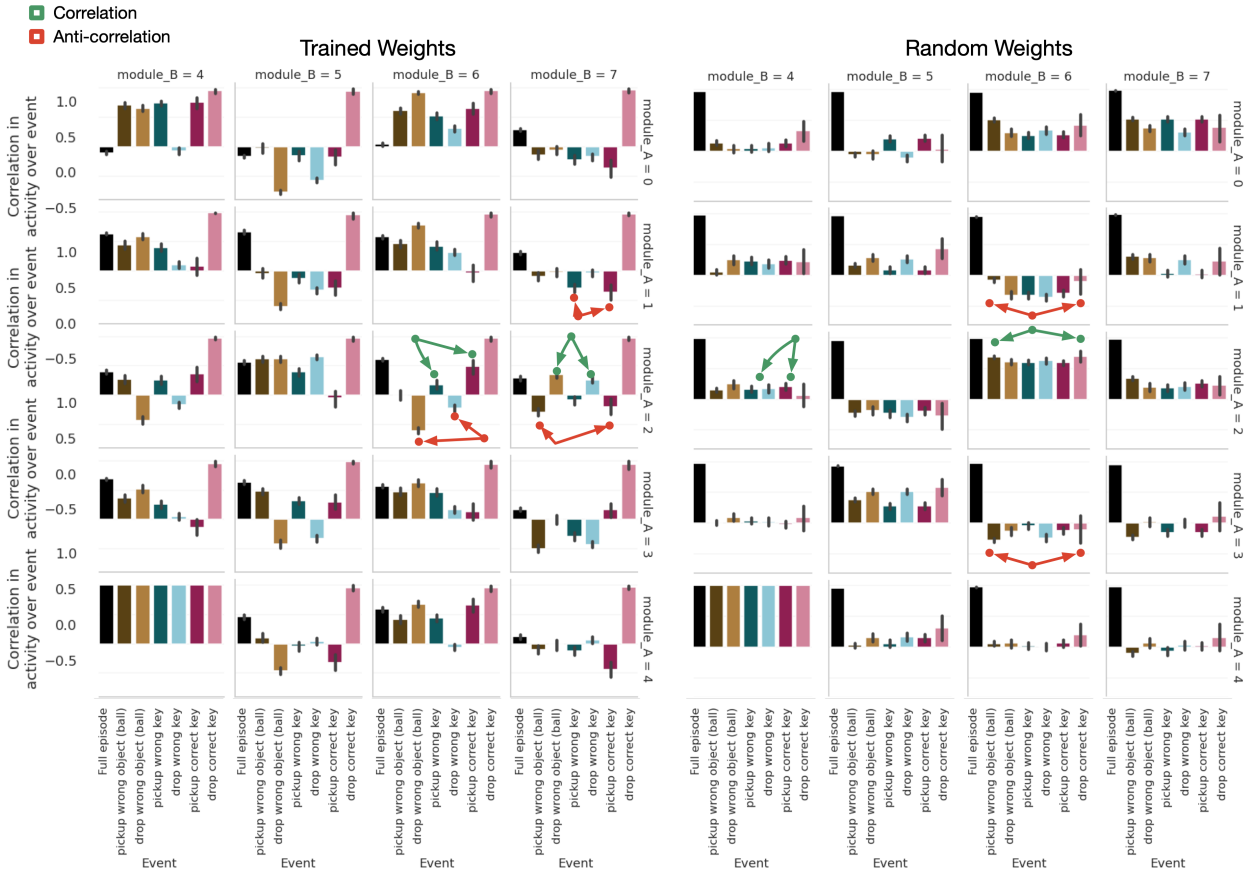

Figure 11: When looking at trained weights (left), we find that pairs of modules will have high correlation on some events and high anti-correlation on other events. For example, modules 7 and 2 correlate for drop wrong object and drop wrong key but anti-correlate pickup wrong object and pickup correct key. If we look at random weights (right), we see that pairs of modules will either fully correlate (modules 6 and 2), fully anti-correlate (modules 6 and 1), or have weak/no correlation (modules 6 and 4) for events. Importantly, we don't see a significant mixture correlation and anti-correlation like we see with trained weights. This suggests that the random weights have less task-specific learning/uses by the agent.

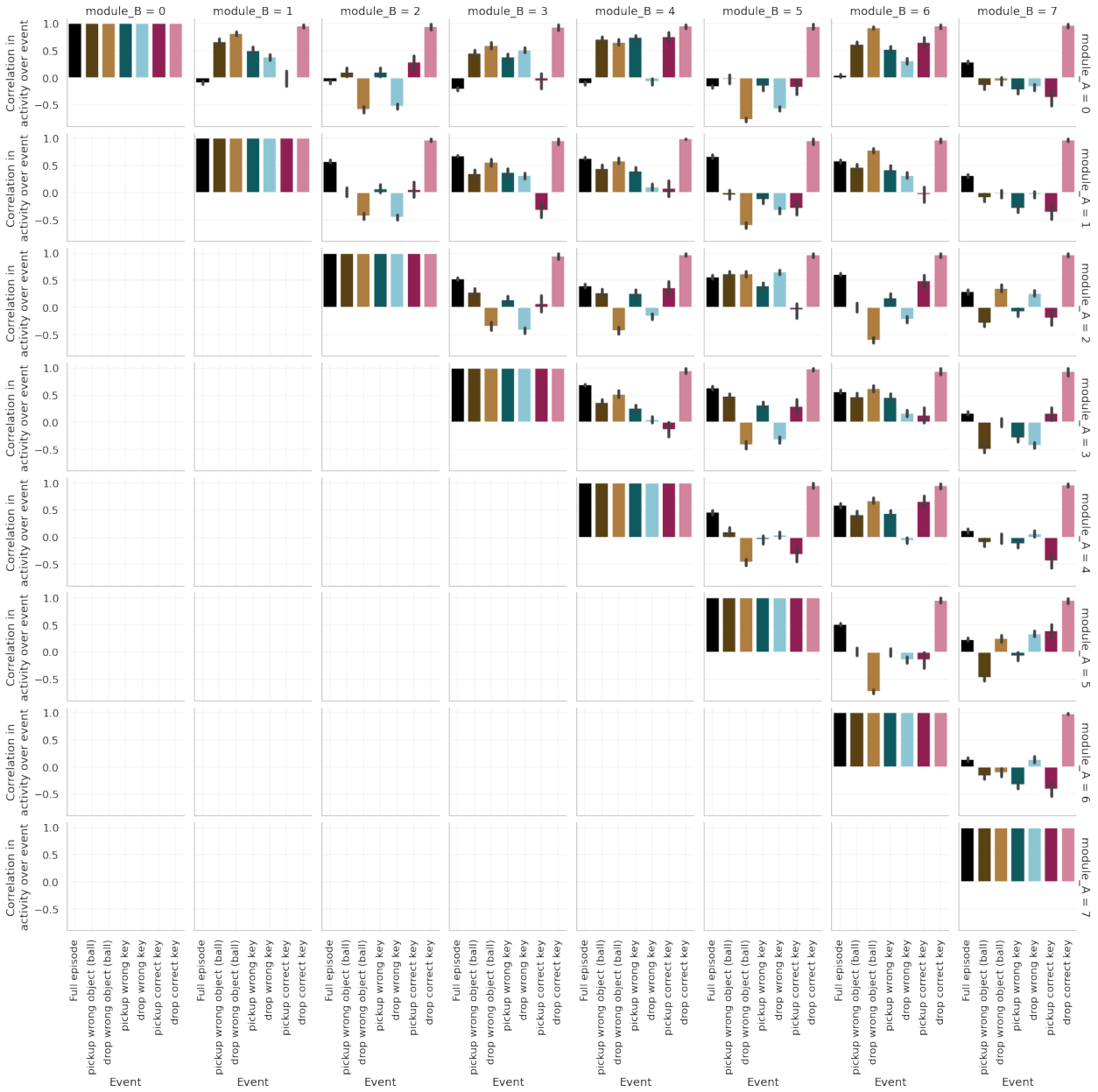

Figure 12: Average pair-wise correlation between L2 norm of module-states.

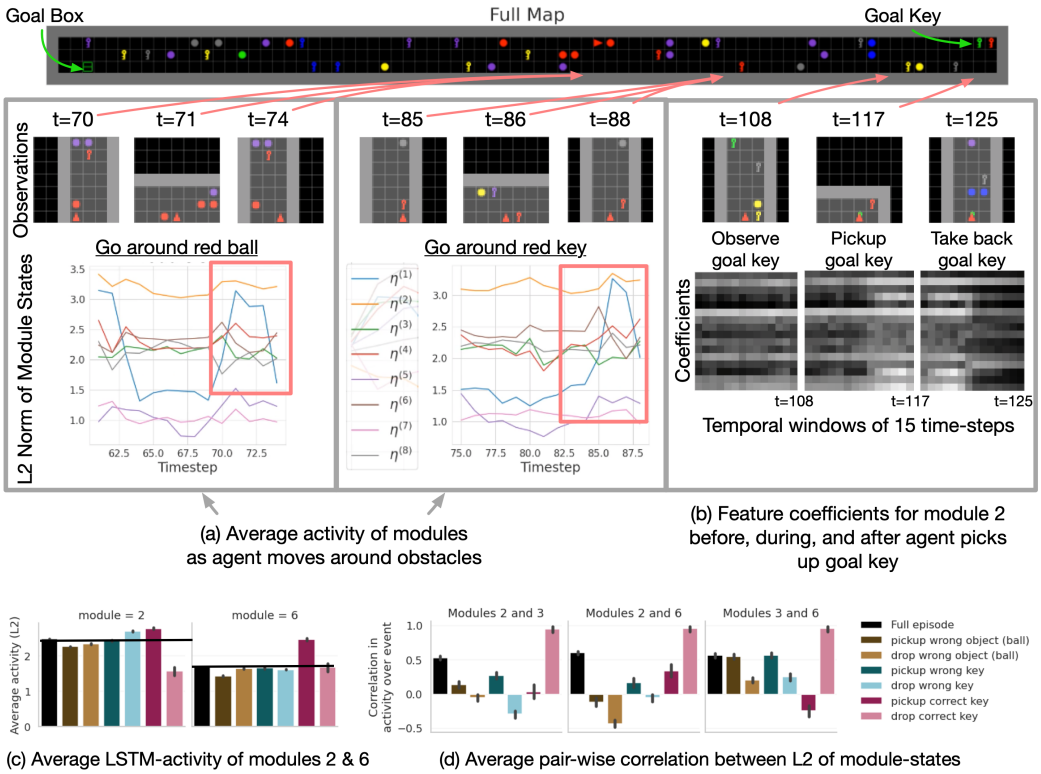

Figure 13: Panels (a) and (b) both show that different modules have selective activity on different events. (a) Module 0 exhibits salient activity when the agent moves around an obstacle. (b) Module 6 shows selective activity for representing goal information. (c) Module 6 also shifts its attention coefficients as the agent picks up the goal key. (d) We generally find that multiple modules activate for an event. Here, modules 3 and 6 show correlated activity for picking up a ball or non-goal key. Videos of the state-activity and attention coefficients over test episodes: https://bit.ly/3qCxatr.

B Additional Experiments

B.1 Generalizing memory-retention to novel spatial compositions of object-dynamics

We use a variant of the the task in §5.1. The main difference is that in this setting, the dancers dance in parallel as opposed to in sequence. This task is no longer a test of memory but only a test of whether the agent can recognize separate object-motions.

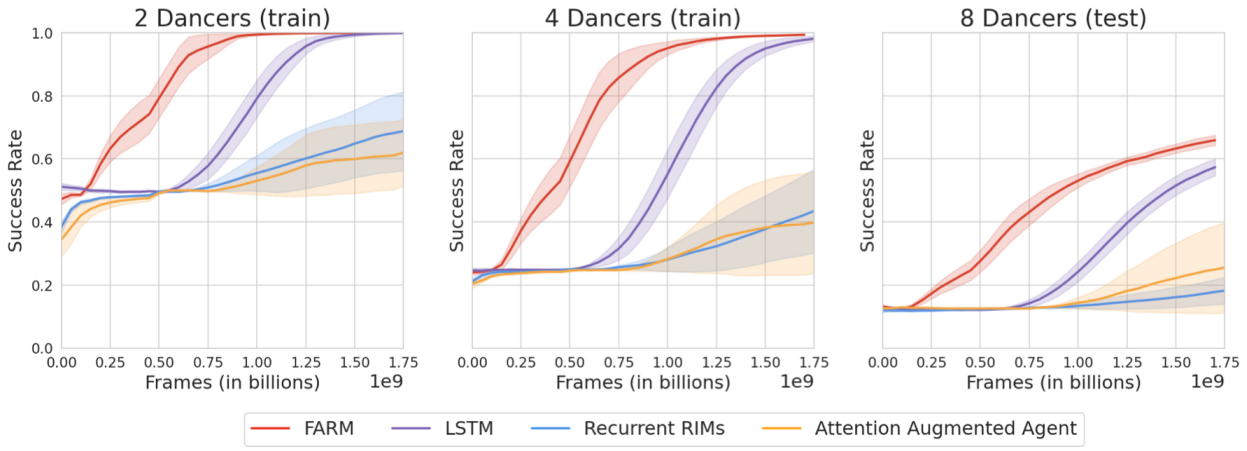

Figure 14: We present the success rate means and standard errors computed using 5 seeds. We find that FARM more quickly learns and generalizes. The next best performance comes from using an LSTM. These results indicate that using spatial attention is an impediment to learning to recognize object-motions.

We present results in Figure 15. In the parallel dancing setting, we find that only FARM and the LSTM can learn these tasks efficiently. Both baselines that use spatial attention learn more slowly and with higher variance.

B.2 Generalizing to an unseen number of distractors

We study this with the "Place X next to Y" task in the BabyAI gridworld (Chevalier-Boisvert et al., 2019) (Figure 18). The agent is a red triangle. Other objects can be squares, boxes or circles and they can take on 7 colors. The agent receives a partial, egocentric observation of the environment (Figure 18, right) and is given a synthetic language instruction. The agent gets a reward of 1 if chooses the correct dancer, and 0 otherwise. During training the agent sees either 0 or 2 distractors. During testing, the agent sees 11 distractors. As the number of distractors increases, the likelihood a distractor is either (a) confounding with the task objects or (b) blocks/confuses the agent also increases.

We present results in Figure 15. On the left two panels, we present training results for $\{0, 2\}$ distractors. All architectures can learn this task. On the right-most panel, we present test results for 11 distractors. FARMand an LSTM get comparable performance ($\approx 70\%$). RIMs has the best generalization success rate ($\approx 80\%$).

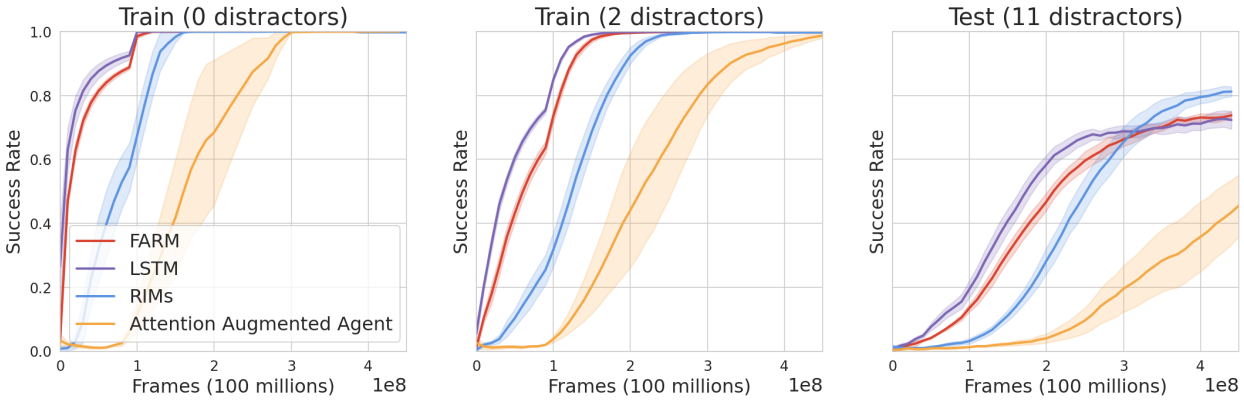

Figure 15: **RIMs, which uses spatial attention, better generalizes to more distractors.** We show train and test success rate performance for "Place X next to Y" in the BabyAI environment (10 runs).

C Unified description of baseline methods

We present a detailed comparison of baseline methods. In Figure 16, we present a schematic of the general architecture that all methods used. The rest of this section is structured as follows. We first recap the general architecture used in all methods, which was describe in§4. Both RIMs and FARM share their method for having modules share information. We describe this in §C.1.1. In §C.1.2, we describe spatial attention vs. feature attention. In §D, we describe implementation details for these pieces.

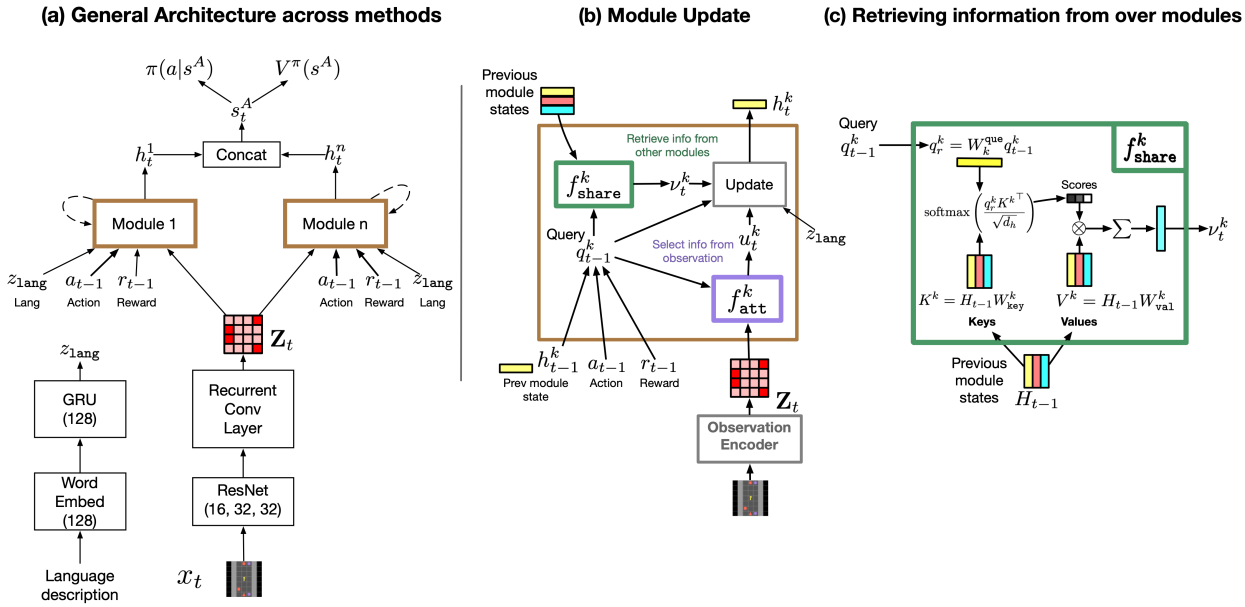

Figure 16: **Schematic of architecture used by all attention-based methods**. The main difference between each method is in (a) the number of modules and (b) the attention function, $f^k_{\texttt{att}}$, used to select information from the observation features. FARM uses n modules and feature attention. RIMs uses n modules and spatial attention. AAA uses 1 module and spatial attention. An LSTM agent uses 1 module that updates with (a) observation features (b) the language encoding, and (c) the previous reward and action.

C.1 General Architecture

At each time-step t, each module updates with both observation features and information from other modules. First, the agent computes observation features with a recurrent observation encoder, $\boldsymbol{Z}_t = \phi(o_t, \boldsymbol{Z}_{t-1})$. Afterward, each module creates a *query* vector by combining its previous module-state with the previous action and reward, $q^k_{t-1} = [h^k_{t-1}, a_{t-1}, r_{t-1}]$. The query is used to attend to observation features via a dynamic feature attention mechanism $u^k_t = f^k_{\texttt{att}}(\boldsymbol{Z}_t, q^k_{t-1})$. The query is also used to retrieve information from other modules with a transformer-style attention mechanism $\nu^k_t = f^k_{\texttt{share}}(s^A_{t-1}, q^k_{t-1})$. (We explain both attention mechanisms in more detail below). Each module updates with both attention outputs to produce the next module-state $h^k_t = \eta^k(u^k_t, \nu^k_t, q^k_{t-1})$. If a task additionally has a language description $o_{\texttt{lang}}$ (as 2 of our experiments do), the module update also updates with an embedding of this description, $z_{\texttt{lang}} = f_{\texttt{lang}}(o_{\texttt{lang}})$. Agent state is then defined by the combination of these module-states $s^A_t = [h^1_t, \ldots, h^n_t]$. We summarize the

computations below:

$$\boldsymbol{Z}_t = \phi(o_t, \boldsymbol{Z}_{t-1}) \quad \text{obs features} \tag{9}$$
$$q^k_{t-1} = [h^k_{t-1}, a_{t-1}, r_{t-1}] \quad \text{query} \tag{10}$$
$$u^k_t = f^k_{\texttt{att}}(\boldsymbol{Z}_t, q^k_{t-1}) \quad \text{obs attention} \tag{11}$$
$$\nu^k_t = f^k_{\texttt{share}}(s^A_{t-1}, q^k_{t-1}) \quad \text{share info} \tag{12}$$
$$h^k_t = \eta^k(u^k_t, \nu^k_t, q^k_{t-1}, z_{\texttt{lang}}) \quad \text{module update} \tag{13}$$
$$s^A_t = [h^1_t, \ldots, h^n_t] \quad \text{agent state} \tag{14}$$

where $[\cdot]$ is an operation that concatenates input vectors into a long vector.

C.1.1 Sharing information ($f^k_{\texttt{share}}$)

Both FARM and RIMs have modules that retrieves information from other modules using transformer-style attention (Vaswani et al., 2017). We define the collection of previous module-states as $\boldsymbol{H}_{t-1} = \left[h^{(1)}_{t-1}; \ldots; h^{(n)}_{t-1}; \boldsymbol{0}\right] \in \mathbb{R}^{(n+1)\times d_h}$, where $\boldsymbol{0}$ is a null-vector used to retrieve no information. A module computes a "retrieval query" to search for information as $q^k_r = W^{\texttt{que}}_k q^k_{t-1} \in R^{d_h}$. That module computes "retrieval keys and values" as $K^k = \boldsymbol{H}_{t-1} W^{\texttt{key}}_k \in R^{n+1\times d_h}$ and $V^k = \boldsymbol{H}_{t-1} W^{\texttt{val}}_k \in R^{n+1\times d_h}$, respectively. Each module then retrieves information as follows:

$$f^k_{\texttt{share}}(s^A_{t-1}, q^k_{t-1}) = \text{softmax}\left(\frac{q^k_r {K^k}^\top}{\sqrt{d_h}}\right) V^k. \tag{15}$$

Intuitively, the dot-product inside the softmax is computing $n+1$ scores (one for each "key"), which then form probabilities. The outter dot-product multiplies each "value" by its probability and sums them to perform soft-selection.

C.1.2 Observation attention ($f^k_{\texttt{attn}}$)

We present a diagram of feature attention vs. spatial attention in Figure 17. Below we describe updating with each type of attention. FARM uses feature attention. RIMs and AAA both use spatial attention.

Updating with Feature Attention. Here, state factors update with "important" features. We focused on visual features produced by a CNN, so this corresponds to important convolutional channels. This method essentially works by applying a learned mask to the convolutional features before updating with them.

Module k transforms its query to a feature mask α^k by projecting the query and applying a sigmoid:

$$\alpha^k = \sigma(W^{\texttt{att}}_k q^k_{t-1})\mathbb{R}^{d_z} \tag{16}$$

where σ is a sigmoid function. Each dimension of the query is bounded between 0 and 1. This essentially gives an importance for updating with each of the d_z features. It then applies this mask to a projection of the convolutional features and then projects the masked features:

$$u^k_t = W_2(\alpha^k \odot W_1 \boldsymbol{Z}_t) \in \mathbb{R}^{m\times d_z} \tag{17}$$

Updating with Spatial Attention (used by RIMs and AAA). Here, state factors update with spatial positions that contain relevant information. A module computes a "spatial query" to search for observation information as

$$q^k_{\texttt{pos}} = W^{\texttt{pos}}_k q^k_{t-1} \in R^{d_q} \tag{18}$$

Observation features are then transformed to "keys" and "values" as $\boldsymbol{Z}^{\texttt{key}} = W^{\texttt{key}}_k \boldsymbol{Z}_t \in \mathbb{R}^{d_q\times m}$ and $\boldsymbol{Z}^{\texttt{val}} = W^{\texttt{val}}_k \boldsymbol{Z}_t \in \mathbb{R}^{d_v\times m}$ (one for each spatial position). Each key is compared against the query, and the best match will be selected. First, "soft" selection scores α^k for each position are computed:

$$\alpha^k = \text{softmax}\left(\frac{q^k_{\texttt{pos}} \boldsymbol{Z}^{\texttt{key}}}{\sqrt{d_q}}\right) \in \mathbb{R}^m \tag{19}$$

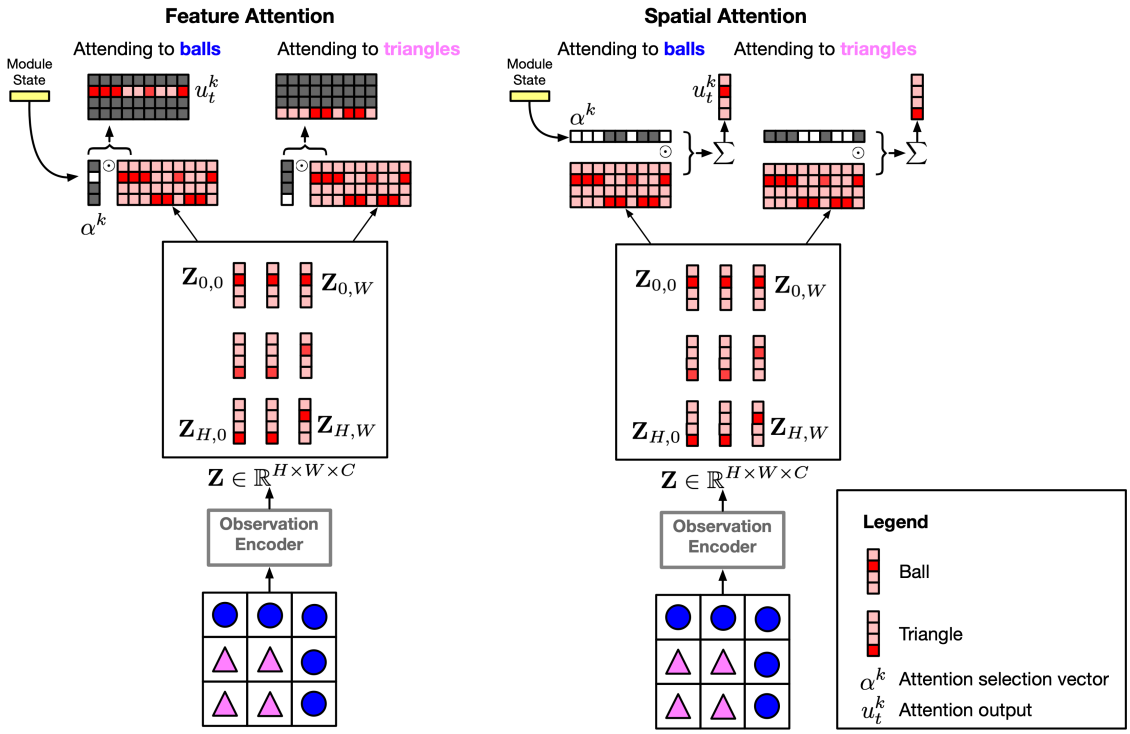

Figure 17: **Feature Attention vs. Spatial Attention**. We present a toy illustration of how modules may attend to different task-relevant objects. Our experiments indicate that modules respond to more abstract features (such as the presence of a general "obstacle" as in Figure 7). Still, it is illustrative to imagine how one could use spatial or feature attention to attend to either balls or triangles. When using feature attention, we get out a matrix that represents where balls or triangles are located across all positions. However, if we use spatial attention, features across spatial positions are averaged together. While the presence of the objects can be determined, their spatial location can be lost. While work can mitigate this by adding positional encodings, the resultant sum will have an average of relevant positional encodings. For the balls, the average of positional encoding may be a location that does not actually contain balls.

One then obtains an update by doing a weighted sum over the values:

$$u_t^k = \sum_{i=1}^{m} \alpha_i^k \boldsymbol{Z}_{:,i}^{\texttt{val}} \tag{20}$$

D Implementation details

All neural networks were built using the Jax library (Bradbury et al., 2018), haiku library (**?**), optax library (**?**), and RLAX RL library. In all experiments, training was carried out using a distributed A3C setup (Espeholt et al., 2018) with discrete actions. We trained all architectures end-to-end with the reinforcement learning objective via the IMPALA algorithm (Espeholt et al., 2018) and an Adam optimizer (Kingma & Ba, 2015). For 3D Unity Env experiments, we added an additional Pixel Control loss (Jaderberg et al., 2016) for all agents. We used a single learner and 256 actors.

Observation encoder. We implement an agent's observation encoders, ϕ, with a ResNet (He et al., 2016). If the observation encoder is recurrent (as with FARM and AAA), the ResNet is followed by a Convolutional LSTM (ConvLSTM) (Shi et al., 2015). **Language encoder**. Language descriptions are processed as follows. First, tokens are embedded into word embeddings and then they are fed into a GRU. The last token GRU embedding is used as the language description $z_{\texttt{lang}}$. **Module update**. During update, modules (a) select information from other modules (RIMs, FARM) ν_t^k and (b) select observation information to update with

u_t^k. Modules then use an LSTM to update with the concatenation of $(\nu_t^k, u_t^k, a_{t-1}, r_{t-1}, h_{t-1}^k)$, where h_{t-1}^k is the modules' previous state. **Sharing information**. For both FARM and RIMs, we use used multihead-attention (Vaswani et al., 2017) for sharing information, $f_{\texttt{share}}^k$ (see column c in Figure 16). For RIMs and AAA, we add positional emebddings for each spatial position of the convolutional features produced by the observation encoder. **RL predictions**. Module states are then concatenated to form the agent's state representation, $s_t^A = [h_t^1, \ldots, h_t^n]$ and used to compute a policy $\pi(a|s_t^A)$ and estimate the state's value $V(s_t^A)$.

E Hyperparamters

Important training hyper-parameters are shown in Table 2, along with the components of the agent's architecture that are shared between the different models. The parameter values used for each model presented in the main paper are shown below in Table 3.

Most hyperparameters (i.e. for our RL algorithm, optimizer, and visual encoder) were tuned using a "vanilla" IMPALA agent that updated state using an LSTM. This is because all methods leveraged an LSTM to update state and we wanted to avoid bias towards our architecture. The only difference is that in AAA, there is one LSTM updating state and in RIMs and FARM, there are multiple LSTMs which are simultaneously being updated.

E.1 Search on gridworld domains

Vanilla IMPALA LSTM agent. We first searched RL algorithm (IMAPALA) and optimizer (Adam) hyperparameters with an LSTM on the "Place X next to Y" BabyAI task (Chevalier-Boisvert et al., 2019). We chose this task because our target domains were object-centric gridworlds and this simple object-centric grid-world acted as a sanity check that our methods worked. We began with default values from our libraries and performed a random search using the following values: V-trace baseline cost $[1.0, .5, 0.1]$, V-trace entropy cost $[10^{-2}, 10^{-3}, 10^{-4}]$, V-trace γ $[1.0, .99, .95]$, Adam learning rate $[10^{-3}, 5 \times 10^{-4}, 2 \times 10^{-3} 10^{-4}]$, LSTM hidden size $[128, 256, 512]$. We consistently found that a larger memory had better results.

Once we found good IMPALA and Adam hyperparameters, we searched over agent-state hyperparameters for each method on the same BabyAI task. **Feature Attending Recurrent Modules**. We searched over attention projection dims $W_i \in [16, 32]$, Conv LSTM kernel size $[3, 5]$, and number of modules $[2, 4, 8]$. We set the ConvLSTM kernel size to be the same size as the final layer of the preceding ResNet. When using multihead attention, the number of attention relation heads is also a hyper-parameter. We fixed this to always be half off the number of modules. We used a per-module LSTM size of 128 and did not vary this across experiments. **Attention Augmented agent**. We used hyper-parameters from their paper but tuned the following: LSTM hidden size $[256, 512]$, Attention query MLP size $[\{\}, \{256\}, \{256, 256\}]$, number of attention heads $[4, 8]$. We consulted the authors about our implementation. **Recurrent Independent Mechanisms**. We used hyper-parameters from their paper but tuned the following: LSTM hidden size $[100, 128, 256]$, Observation/communication head size $[32, 64, 128]$, number of observation/communication heads $[4, 5, 6]$, number of RIMs $[4, 6, 9, 12]$. We consulted the authors about our implementation and used their source code for replication.

Finally, once we had good hyperparameters for agent-state, we applied the architectures to the "Ballet" and "Keybox" gridworld domains and explored whether increasing agent-state capacity (e.g. LSTM size or number of LSTMs) improved performance. We tried combinations of LSTM size and number of LSTMs that led each method to have approximately the same number of parameters. This was to ensure that no method performed better than the other simply because it had more parameters.

E.2 Search on 3D unity domain

We recompleted our initial search on the RL algorithm (IMAPALA) and optimizer (Adam) hyperparameters. We searched over the same values as before and additionally searched over a larger MLPs for the policy and value heads $[200, 512]$, Adam optimizer episilon $[10^{-7}, 5 \times 10^{-8}, 10^{-8}]$, Adam $\beta_1[0.0, .9, .95, .99, .999]$ and $\beta_2[0.0, .9, .95, .99, .999]$, and did a small search over the Pixel Control loss scaling $[0.1, 0.01, 0.001]$ and Pixel Control discount factor $[0.9, .99]$. After we searched the IMPALA, Adam, and Pixel Control hyperparameters, we searched over individual architecture hyperparameters again.

Table 2: Training hyper-parameters and shared network components used in experiments.

Loss Hyper-parameters	**3D Unity Env**	**Gridworlds**
V-trace baseline cost	1.0	0.5
V-trace entropy cost	10^{-4}	0.01
V-trace γ	0.95	1.0
V-trace loss scaling	0.1	1.0
Pixel Control loss scaling	0.1	–
Pixel Control loss cell size	4	–
Pixel Control discount factor	0.9	–
Optimizer	clipped Adam	clipped Adam
Learning rate	2×10^{-4}	10^{-4}
Max gradient Norm	40.0	40.0
Optimizer epsilon	5×10^{-8}	10^{-8}
Adam β_1	0.0	0.9
Adam β_2	0.95	0.999
Shared Network Components		
Language encoder	GRU	GUR
Language encoder hidden sizes	128	128
Language word embedding size	128	128
Image encoder	Res-Net	Res-Net
Res-Net channels	(16, 32, 32)	(16, 32, 32)
Res-Net residual blocks	(2, 2, 2)	(2, 2, 2)
Res-Net stride	2	2
Res-Net kernel size	3	3
Res-Net padding	SAME	SAME
Image-language-reward-action combination	Concatenation	Concatenation
Policy Head MLP shapes	[512, 46]	[200, 7]
Value Head MLP shapes	[512, 1]	[200, 1]

Table 3: Model specific parameters. We highlight values that changed across environments in blue.

Model parameter	**3D Unity Env**	**Gridworld Ballet**	**Gridworld Keybox**
Observation Dims	72×96	99×99	56×56
Feature-Attending Recurrent Modules			
Parameters (millions)	5.1	7.1	7.6
Number of modules	4	4	8
Module-state LSTM size	128	128	128
$\frac{\text{Relation heads}}{\text{Number of modules}}$	.5	.5	.5
Projection dims W_1, W_2	16	16	16
ConvLSTM kernel size	3	3	3
ConvLSTM hidden size	32	32	32
LSTM			
Parameters (millions)	5.6	7.2	7.6
LSTM size	896	768	1024
Attention Augmented Agent			
Parameters (millions)	5.1	6.9	7.5
ConvLSTM kernel size	3	3	3
ConvLSTM output size	128	128	128
LSTM size	704	512	960
Number of attention heads	4	4	4
Attention query MLP size	(256, 256)	(256, 256)	(256, 256)
Positional basis dim	4	4	4
RIMs			
Parameters (millions)	5	6.6	7.6
Number of modules	12	9	9
LSTM size	128	128	128
Observation heads	6	6	6
Communication heads	6	6	6
Observation head size	32	32	32
Communication head size	32	32	32
Basis size	4	4	4
Dropout	0.2	0.2	0.2

F Environments

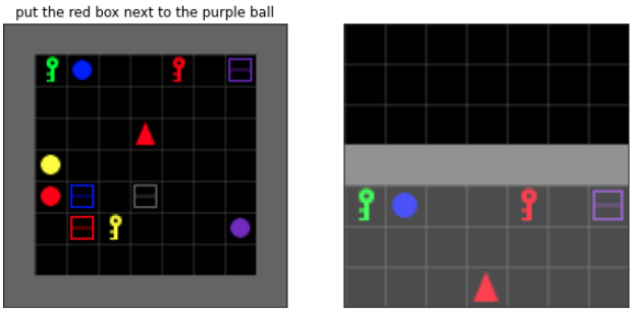

Figure 18: Place X on Y task in BabyAI environment.

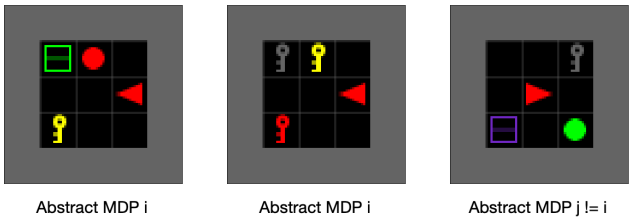

Figure 19: Abstract MDP Environment based on BabyAI.

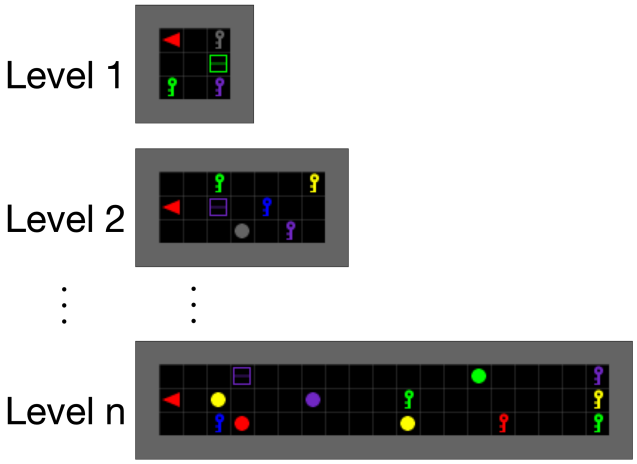

Figure 20: KeyBox task.

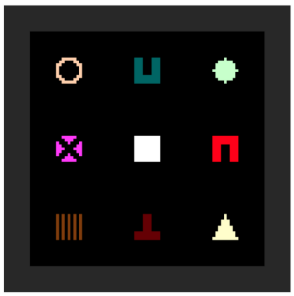

Figure 21: Ballet task.

Figure 22: Additional Environment Images.

F.1 Ballet

Please refer to Lampinen et al. (2021) for details on this task. Our only difference was to use tasks with $\{2, 4\}$ dancers during training and tasks with 8 dancers for testing.

F.2 KeyBox

Observation Space. The agent receives a 56×56 partially observable, egocentric image of the environment as in Figure 18, right.

Action Space. The action space is composed of the 7 discrete actions *turn left*, *turn right*, *go forward*, *pickup object*, *drop object*, *toggle*, and *done/no-op*.

Reward function. When the agent completes level n, it gets a reward of $n/n_{\texttt{max}}$ where $n_{\texttt{max}}$ is the maximum level the agent can complete. We set $n_{\texttt{max}} = 10$ during training. The agent has $50n$ time-steps to complete a level.

Table 4: Object and colors available for objects in the KeyBox task.

Set	Contains
Shapes	ball, key, box
Colors	red, green, blue, purple, pink, yellow, white

F.3 3d Unity Environment

For the "place X on Y" experiments in 3D, all pickupable objects were split into two sets $O_1 = A \cup B$ and all object to place something on into another two sets $O_2 = C \cup D$, as shown in Table 5. Given the challenging nature of the 3D environment (huge number of possible states, partial observability, language commands, long credit assignment), we had to employ a set of curriculum tasks in order for the agents to make any progress on the actual task of interest "Put X on Y". The agent co-trained on the full set of tasks. This was possible since we used a distributed A3C setup for our training (Espeholt et al., 2018), where each of the actors generating the experience was running on one of the possible training levels. The different training tasks used during training and evaluation are shown in Table 6.

All episodes lasted for a maximum of 120 seconds and an action repeat of 4 was used. The images observations were rendered at $96 \times 72 \times 3$ and given to the agent along with a text language instruction, where each word in the instruction was mapped into a continuous vector of size 128 using a fixed vocabulary of maximum size 1000.

Reward function. An agent get's a reward of 1 if it completed the task and 0 otherwise.

Action Space. The action space for the experiments in 3d Unity Environment was 46 discrete actions that allow the agent to move its body and change its head direction, to grab objects while moving and manipulate the held objects by rotating, pulling or pushing the held object. The object is while as long as the agent is emitting a GRAB action, and dropped in the first instance that a GRAB action is not emitted. The full list of possible actions in the 3d Unity Environment environment is presented in Table 7.

Table 5: Object and color set splits for the 3d Unity Environment "Put X on Y" experiments.

Set	Contains
Set A (pickupable objects)	toilet roll, toothbrush, toothpaste
Set B (pickupable objects)	bus, car, carriage, helicopter, keyboard
Set C (support object)	stool, tv cabinet, wardrobe, wash basin
Set D (support object)	bed, book case, chest, dining, table
Colors	red, green, blue, aquamarine, magenta, orange, purple, pink, yellow, white

Table 6: Descriptions of all the tasks used during training and evaluation. D refers to number of distractors and S to the room size.

Task name	S	D	Description
Find X (Set A or B)	4×4	5	The agent is spawned randomly. Room has 3 objects from Set A (or B) and 3 from $C \cup D$ and instructed to go to an object from Set A (or B). The purpose of these training tasks is to associate objects from Set A and B with their names and the "find" instruction with finding them.
Find Y (Set $C \cup D$)	4×4	5	The agent is spawned randomly. Room has 3 objects from Set A (or B) and 3 from $C \cup D$ and instructed to go to an object from Set $C \cup D$. The purpose of these training tasks is to associate objects from Set $C \cup D$ with their names and the "find" instruction with finding them.
Lift X (Set A or B)	4×4	5	The agent is spawned randomly. Room has 3 objects from Set A (or B) and 3 from $C \cup D$ and instructed to lift an object from Set A (or B). The purpose of these training tasks is to associate the "lift" instruction with lifting the said object.
Put X near Y (X = Set A or B, Y = Set $C \cup D$)	3×3	0	The agent is spawned randomly. Room has 1 object from Set A (or B) and 1 from $C \cup D$ and instructed to put the object from Set A (or B) near the other. The purpose of these training tasks is to learn to move one object near another before putting it on it.
Put X on Y (X = Set A or B, Y = Set $C \cup D$)	3×3	0	The agent is spawned randomly. Room has 1 object from Set A (or B) and 1 from $C \cup D$ and instructed to put the object from Set A (or B) on top of the other. The purpose of these training tasks is to learn to move one object and place it on top of another.
Put X on Y (X = A, Y = D or X = B, Y = C)	4×4	4	The agent is spawned randomly. Room has 3 objects from Set A (or B) and 3 from Set D (or C) and instructed to put the object from Set A (or B) on top of the other. This is the training task most similar to the test task and requires mastering all the other ones.
Put X on Y **(test)** (X = A, Y = C or X = B, Y = D)	4×4	4	The agent is spawned randomly. Room has 3 objects from Set A (or B) and 3 from Set C (or D) and instructed to put the object from Set A (or B) on top of the other. This is the test task.

Table 7: 3d Unity Environment action space.

General body movement	**Fine grain movement**
NOOP	MOVE_RIGHT_SLIGHTLY
MOVE_FORWARD_FULL	MOVE_LEFT_SLIGHTLY
MOVE_BACKWARD_FULL	LOOK_RIGHT_MID
MOVE_RIGHT_FULL	LOOK_LEFT_MID
MOVE_LEFT_FULL	LOOK_DOWN_MID
LOOK_RIGHT_FULL	LOOK_UP_MID
LOOK_LEFT_FULL	LOOK_RIGHT_SLIGHTLY
LOOK_DOWN_FULL	LOOK_LEFT_SLIGHTLY
LOOK_UP_FULL	
Fine grained movement with grip	**General body movement with grip**
GRAB + MOVE_RIGHT_MID	GRAB
GRAB + MOVE_LEFT_MID	GRAB + MOVE_FORWARD_FULL
GRAB + LOOK_RIGHT_MID	GRAB + MOVE_BACKWARD_FULL
GRAB + LOOK_LEFT_MID	GRAB + MOVE_RIGHT_FULL
GRAB + LOOK_DOWN_MID	GRAB + MOVE_LEFT_FULL
GRAB + LOOK_UP_MID	GRAB + LOOK_RIGHT_FULL
GRAB + LOOK_RIGHT_SLIGHTLY	GRAB + LOOK_LEFT_FULL
GRAB + LOOK_LEFT_SLIGHTLY	GRAB + LOOK_DOWN_FULL
GRAB + PULL_CLOSER_MID	GRAB + LOOK_UP_FULL
GRAB + PUSH_AWAY_MID	
Object manipulation	
GRAB + SPIN_RIGHT	
GRAB + SPIN_LEFT	
GRAB + SPIN_UP	
GRAB + SPIN_DOWN	
GRAB + SPIN_FORWARD	
GRAB + SPIN_BACKWARD	
GRAB + PULL_CLOSER_FULL	
GRAB + PUSH_AWAY_FULL	
PULL_CLOSER_MID	
PUSH_AWAY_MID	