Supplementary material for CompILE: Compositional Imitation Learning and Execution

A. CompILE model details

A.1. Encoder architecture

Grid world Both the recognition model and the generative model (i.e., the sub-task policies) use a two-layer CNN with 3×3 filters and 64 feature maps in each layer, followed by a ReLU activation each. We flatten the output representation into a vector and pass it through another trainable linear layer, without activation function. Only for the recognition model, we further concatenate a linear (trainable) embedding of the action ID to this representation. In all cases, we pass the output through a LayerNorm (Ba et al., 2016) layer before it is passed on to other parts of the model, e.g. the RNN in the recognition model or the sub-task policy MLP in the generative model.

The LSTM state of the recognition model is reset to 0 between trajectories (and after each pass over the trajectory, i.e., for each segment).

Continuous control environment The model architecture for the continuous control environment is the same as the grid world, except that the input encoder uses MLPs of 2 hidden layers of 256 units each with ReLU activations, instead of CNNs.

A.2. Sub-task policies

The sub-task policies $\pi_\theta(a|s,z)$ are composed of a CNN module to embed the environment state s_t and a subsequent MLP head to predict the probability of taking a particular action. This CNN shares the same architecture as the recognition model CNN. In initial experiments, we found that training separate policies $\pi_{\theta_z}(a|s)$ for each sub-task $z \in \{1, ..., K\}$ with shared CNN parameters led to better generalization performance than embedding the sub-task latent variable and providing it as input to just a single policy for all sub-tasks. For continuously relaxed latent variables z, i.e. during training, we use a soft mixture $\pi_\theta(a|s,z) = \sum_{k=1:K} q(z = k|a,s,b)\pi_{\theta_k}(a|s)$ to obtain gradients, where we have omitted time step and segment indices to simplify notation.

A.3. Termination policy

To allow for our model to be used in an online setting where the end of an event segment has to be identified before "seeing the future", we jointly train a termination policy that shares the same model architecture (but without shared parameters) as the boundary prediction network $q_{\phi_b}(b_i|x)$, but with a $\mathrm{sigmoid}(x) = 1/(1+e^{-x})$ activation function on the logits instead of a (Gumbel) softmax. It similarly passes over the input sequence M times (with softly masked out RNN hidden states) and is trained to predict an output of 1 (i.e., terminate) for the location of the i-th boundary $b_i = \mathrm{argmax}_{t=1:T} q_{\phi_b}(b_i = t|x)$ and zero otherwise. At test time, we use a threshold of 0.5 to determine termination.

A.4. ELBO objective for learning

We jointly optimize for both the parameters of the sub-task policy $\pi_\theta(a|s,z)$ and the recognition model $q_\phi(b,z|a,s)$ by using the ELBO as an objective for learning:

$$\begin{aligned}\mathrm{ELBO} = \mathbb{E}_{q_\phi(b,z|a,s)}[&\log p_\theta(a|s,b,z)\\ &+ \log p(b,z) - \log q_\phi(b,z|a,s)],\end{aligned} \tag{1}$$

where we have dropped time step and sub-task indices for ease of notation. The first term can be understood as the (negative) reconstruction error of the action sequence, given a sequence of states and inferred latent variables, whereas the last two terms, in expectation, form the Kullback-Leibler (KL) divergence between the prior $p(b,z)$ and the posterior $q_\phi(b,z|a,s)$. The ELBO can be obtained from the original BC objective as follows, using Jensen's inequality:

$$\begin{aligned}\log p_\theta(a|s) &= \log \sum_{b,z} p_\theta(a|s,b,z)p(b,z)\\ &= \log\, \mathbb{E}_{q_\phi(b,z|a,s)}\left[\frac{p_\theta(a|s,b,z)p(b,z)}{q_\phi(b,z|a,s)}\right]\\ &\geq \mathbb{E}_{q_\phi(b,z|a,s)}\left[\log\frac{p_\theta(a|s,b,z)p(b,z)}{q_\phi(b,z|a,s)}\right]\\ &= \mathrm{ELBO}\end{aligned} \tag{2}$$

An overview of the dependencies between observed and latent variables in our model is provided in Figure 1.

A.5. KL term

We use a scale hyperparameter $\beta \in [0,1]$ to scale the contribution of the KL term in Eq. (1) similar to the β-VAE framework (Higgins et al., 2017), which gives us control over the strength of the prior $p(b,z)$. As is common in applications of relaxed categorical posteriors in a VAE (Jang

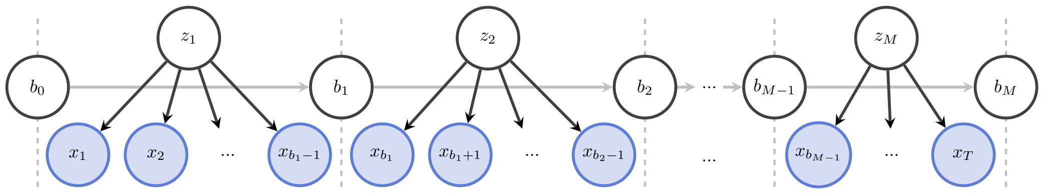

Figure 1: Dependencies between observed and latent variables in our generative model $p_\theta(x_{1:T}|b_{1:M}, z_{1:M})$. The state-action pair (a_t, s_t) is summarized into a single observed variable x_t. The latent variables b_i determine the location of the boundaries between segments, whereas z_i summarize the content of each segment.

et al., 2017), we choose a simple (non-relaxed) categorical KL term for both the posterior distributions $q_{\phi_b}(b_i|x)$ and $q_{\phi_z}(z_i|x)$.

Further, as we do not know the precise location of the boundary latent variables b_i at training time, we cannot evaluate $p(b_i|b_{i-1})$ for $i > 1$ in the relaxed/continuous case. Under the assumption of independence between segments, behavior within each segment originating from the same distribution, and with a shared recognition model for all latents, we can equivalently evaluate the KL term related to b for the first boundary only, i.e. for $p(b_1)$, and multiply this term by M, where M is the number of segments (we use this setting in our experiments). Alternatively, one could place a prior on $\sum_{t=1:T} P(t \in C_i)$, which can be understood as a continuous relaxation of the length of a segment. This would allow for an individual KL contribution for every segment, which could be useful for other applications or environments, where our assumptions are too restrictive.

A.6. Gaussian latent variables

We experimented with continuous, Gaussian latent variables z in the grid world domain and found that our model can support this setting with only minor modifications. We use a single policy $\pi_\theta(a|s, z)$ for decoding, where the MLP head takes the latent variable z (passed through a single, trainable linear layer) as input in addition to the state embedding (both are concatenated). We further place a unit-variance, zero-mean Gaussian prior on z and use the appropriate KL term. We trained and tested this model variant under the same setting as the experiments with discrete latent variables, with the exception of using 32-dimensional Gaussian latent variables. Results for this setting are summarized in Figure 2 for the grid world domain and in Table 1 for the continuous control domain.

Model	Accuracy	F1 (tol=0)	F1 (tol=1)
3 tasks			
LSTM surprisal	24.8 ± 0.6	39.0 ± 0.3	47.1 ± 0.4
CompILE	45.2 ± 13.8	59.3 ± 12.1	68.8 ± 8.1
z-CompILE	99.6 ± 0.2	99.6 ± 0.1	99.9 ± 0.1
b-CompILE	99.8 ± 0.1	99.9 ± 0.1	99.9 ± 0.0
5 tasks – generalization			
LSTM surprisal	21.6 ± 0.5	44.9 ± 0.5	54.4 ± 0.5
CompILE	28.7 ± 7.0	56.4 ± 9.1	63.8 ± 6.5
z-CompILE	98.3 ± 0.5	99.2 ± 0.3	99.7 ± 0.1
b-CompILE	98.5 ± 0.3	99.3 ± 0.2	99.7 ± 0.1

Table 1: Segmentation results in continuous control domain for CompILE model variant with Gaussian latent variables. Values are in % and we report mean and standard deviation for runs with 5 different random seeds.

A.7. Attentive readout

Instead of (softly) reading the logits for the latent variables z_i from the last time step within a segment, we experimented with using a learned attention mechanism, masked by the respective soft segment mask. In this setting, we add another output head (a single, learnable linear layer) on top of the recognition model RNN which we denote by a_i^t, where t stands for the time step and i denotes the segment index. Before passing the attention scores a_i^t through a softmax layer, we re-normalize using the segment probability $P(t \in C_i)$:

$$\tilde{a}_i^t = a_i^t + \log P(t \in C_i), \tag{3}$$

i.e. we softly mask the attention scores so that the read-out is only performed within the respective segment. The final attention score is obtained as $s_i = \mathrm{softmax}(\tilde{a}_i)$, where the softmax is applied over the time dimension. We read out the logits of z_i from the output heads as follows:

$$q_{\phi_z}(z_i|x) = \mathrm{concrete}_\tau\Big(\sum_{t=1:T} s_i^t h_{z_i}^t\Big). \tag{4}$$

We found that results were similar in both settings and that the model typically learned to attend to the last time step within the segment. For different environments where the cue for a specific sub-goal in a segment of behavior appears at different locations within the segment, the attention mechanism will potentially be a better fit than a soft read-out at the end of the segment.

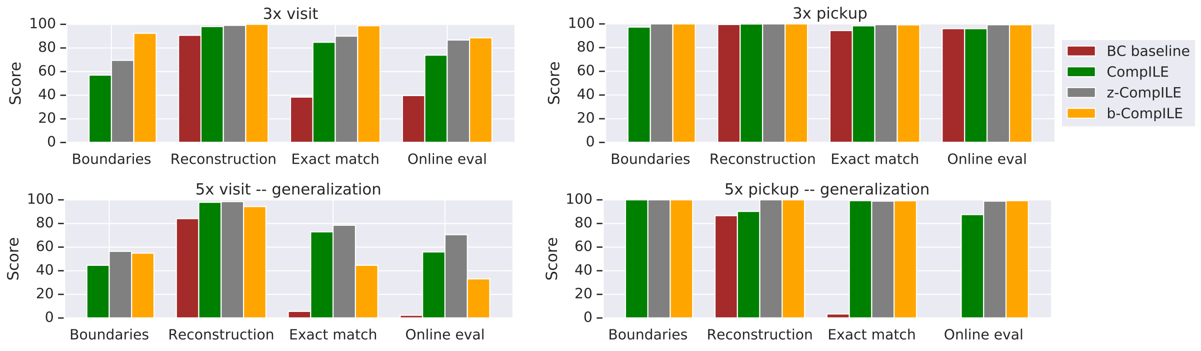

Figure 2: Imitation learning results in grid world domain for CompILE model variant with Gaussian latent variables.

Hyperparam.	Accuracy	F1 (tol=0)	F1 (tol=1)
	# Segments M		
$M = 3$	62.0 ± 4.5	74.3 ± 3.3	78.9 ± 2.5
$M = 4$	53.4 ± 6.3	66.6 ± 5.5	75.3 ± 3.1
$M = 5$	29.8 ± 6.1	47.3 ± 5.5	65.6 ± 2.9
	Softmax temperature τ		
$\tau = 1$	62.0 ± 4.5	74.3 ± 3.3	78.9 ± 2.5
$\tau = 0.1$	40.9 ± 5.3	55.3 ± 5.7	67.0 ± 2.4
$\tau = 0.01$	36.3 ± 6.0	50.3 ± 5.2	66.4 ± 3.6

Table 2: Segmentation results in continuous control domain with 3 tasks for CompILE model trained with (a) different number of segments ranging from $M = 3$ (correct setting) to $M = 5$ (too many boundaries provided to the model), and (b) softmax temperature ranging from $\tau = 1$ to $\tau = 0.01$. Values are in % and we report mean and standard deviation for runs with 5 different random seeds.

A.8. Other hyperparameters

Number of hidden units and MLP layers We use 256 hidden units in all MLP layers and in the LSTM throughout all experiments, unless otherwise mentioned. A smaller number of hidden units mostly did not affect the boundary prediction accuracy, but slightly reduced performance in terms of reconstruction accuracy. For the output heads for h_{z_i}, we use a single, trainable linear layer (we experimented with deeper MLPs but didn't find a difference in performance) and we use a single hidden layer MLP with ReLU activation function for the output head h_{b_i} (the output is a scalar for every time step). Similarly, the policy MLP is using a single hidden layer with ReLU activation in the maze task, while for the control task we used a 2 layer MLP. The termination policy uses an MLP with two hidden layers with ReLU activation functions on top of the RNN outputs.

Number of segments The hyperparameter M, i.e., the number of segments that the model is allowed to use to explain a particular input sequence, can have an impact on reconstruction and segmentation quality. We generally find that we obtain best results by providing the model with the true number of underlying segments (if this number is known). When providing the model with more than necessary segments, it often learns to place unneeded segmentation boundary indicators at the end of the sequence, while in some cases the model over-segments the trajectory (i.e., it breaks a single segment into parts). We provide results for this setting on the continuous control task in Table 2, and we find that the accuracy (and F1 score) for segmentation boundary placement slightly degrades if the model is provided with more than necessary segments.

Softmax temperature We experimented with annealing the Gumbel softmax temperature over the course of training, starting from a temperature of 1 and found that it could slightly improve results, depending on the precise choice of annealing schedule and final temperature. To simplify the exposition and to allow for easier reproduction, however, we report results with fixed temperature of 1 throughout training unless otherwise mentioned. In Table 2, we provide results for experiments with lower softmax temperature (fixed throughout training) on the continuous control task. We found that the boundary prediction accuracy degrades when training with lower temperatures without annealing. When training with partial supervision on either the boundary positions (b-CompILE) or segment encodings (z-CompILE), we found that results are unaffected by lower softmax temperatures.

Poisson prior rate We fix the Poisson rate to $\lambda = 3$ in all experiments. We found that our model was not very sensitive to the precise value of λ.

B. Reinforcement learning agent details

B.1. Architecture and hyperparameters

The agent uses a smaller model than our CompILE imitation learning model, but otherwise similarly has a 2-layer CNN encoder followed by an MLP policy. The CNN has 3×3

filters with 32 feature maps, followed by an MLP with two hidden layers of size 128. Both the CNN and the MLP use ReLU activations. All agents use the same architecture, and the hierarchical agent based on the pre-trained CompILE model uses 128 instead of 256 hidden units (otherwise same training and same architecture as in the imitation learning experiments). The hierarchical agent has access to both low-level actions (8 in total) and 10 meta-actions which correspond to executing one sub-policy of the CompILE model.

The baseline VAE-based BC agent corresponds to an ablation of the hierachical CompILE-based agent, where we use only a single segment (i.e. $M = 1$, no segmentation) during training and a 128-dimensional categorical latent variable z (instead of 10 categories). The agent therefore can choose between 128 meta-actions and 10 low-level actions.

We embed the current task type (visit or pick up) and object type each in a 16-dim vector, via a trainable linear layer. These are concatenated and provided to the policy model in the following two ways: 1) we concatenate this embedding vector with the current observation along the channel (object type) dimension before we feed it into the CNN, and 2) we concatenate the embedding vector with the last hidden layer of the policy MLP. The former allows the CNN to be conditioned on the task type, while we found the second concatenation in the policy MLP to help convergence. For the VAE-based BC baseline (which tries to solve multiple tasks at once), we do not just provide the current task, but the full list of remaining tasks by embedding each task and concatenating them into a single vector (with zero-padding for already fulfilled tasks).

For IMPALA (Espeholt et al., 2018), we use an entropy cost factor of 0.0005, a baseline cost factor of 0.5, and a discounting factor of 0.99. The agents are trained with the Adam optimizer (Kingma & Ba, 2015) using a learning rate of 0.001 and a batch size of 256.

B.2. Distributed training

We distribute the training of this agent into one learner and multiple actors following the IMPALA framework (Espeholt et al., 2018), where the actors generate trajectories using the current agent parameters for training, and the learner updates the agent parameters based on the trajectories received from the actors. The learner runs on a GPU, while the actors run on CPUs. The number of actors is tuned to maximize the throughput of the learner.

This framework uses the actor-critic training algorithm, with off-policy correction (Espeholt et al., 2018) to handle the staleness of the actor generated trajectories. This correction is necessary as the actors and the learner are not always in sync in a distributed setting, and the parameter weights used for generating trajectories are usually not the latest learner weights when the learner receives the trajectories.

C. Environment implementation details

C.1. Grid world

The environment is implemented in pycolab (`https://github.com/deepmind/pycolab`) with 8 different primitive actions: move north, move east, move south, move west, pick up north, pick up east, pick up south, pick up west. Each executed action corresponds to one time step in the environment. Observations s_t are tensors of shape $10 \times 10 \times N_{\text{things}}$, where N_{things} is the total number of things available in the environment, in our case these are 10 object types that can be interacted with, impassable walls and the player, i.e. $N_{\text{things}} = 12$. We ensure that the task is solvable and no walls make objects unreachable. Walls are placed using a recursive backtracking algorithm for unbiased maze generation. We further subsample walls using a sampling rate of 0.2 to simplify the task. The 2D grid is enclosed by a single row/column of walls that are not subsampled.

Demonstration sequences are generated using a breadth-first search on the graph defined by all allowed movement transitions to find the shortest path to the goal object (ties are broken in a consistent manner). For pick up instructions, we replace the last move action in the demonstration sequence with a directional pick up action. We cut demonstration sequences to a maximum length of 42 at training time, and 200 at test time (as some of our tests involve more tasks).

C.2. Continuous control

This environment is adapted from the single target reacher task in DeepMind control suite (Tassa et al., 2018). The reacher arm is composed of two segments, each with length $l = 0.12$, and the controller controls the two motors on the two joints of the arm, one at the shoulder and the other at the elbow. The control actions are the angular velocities to be applied at the two joints. Target objects (spheres) have a diameter of $d = 0.05$, and they are placed in a belt around the center, with the distance to the center sampled uniformly from range $[0.05, 0.2]$, and direction (angle) sampled uniformly around the circle. The environment is set up to take control actions in time intervals of 0.06, with each episode taking a maximum time of 6, i.e. 100 time steps at most.

In this customized environment, we have a total of $K = 10$ distinct target types, each designated with a different color in the rendered scenes. Each target is represented using 3 numbers (α, x, y), where α is the visibility of the object, and $\alpha = 1$ if the object is visible, and $\alpha = 0$ otherwise, (x, y) is the Cartesian coordinate of the target.

In each episode, we first set the number of tasks to $M = 3$

or $M = 5$, and then sample the number of objects N in range $[M, 6]$ uniformly, and then pick M out of N objects uniformly without replacement as the targets to create a task list.

The agent receives an observation that is composed of 2 parts, the first part is the concatenation of all object tuples, arranged in a vector like this: $(\alpha_1, x_1, y_1, \alpha_2, x_2, y_2, ..., \alpha_K, x_K, y_K)$, where (α_i, x_i, y_i) describes the ith object type. If the ith object type is not selected (not among the N objects being selected) in this episode, then all of α_i, x_i and y_i are set to 0. The second part is the position of the reacher arm represented as two angles (θ_1, θ_2), where θ_1 is the angle at the shoulder joint, and θ_2 is the angle at the elbow joint.

The coordinate of the finger tip of the arm is computed as $(l\cos\theta_1 + l\cos(\theta_1+\theta_2), l\sin\theta_1 + l\sin(\theta_1+\theta_2))$. A target is considered reached if this coordinate is within the sphere for the given target.

Once a target is reached, the α value for that target is set to 0 (but the x and y values remain in the observation), and in the next time step the environment advances to the next task, with a new target being selected as the goal.

The demonstration trajectories are generated by a hand-designed controller. The controller has access to the coordinates of the next target. It first computes the coordinates of the finger tip, and then computes (1) the distance of the finger tip to the center (where the shoulder joint is); and (2) the angle of the finger tip. If the distance is smaller than the distance of the target to the center, the elbow motor applies an angular velocity to open the arm (so that the finger tip can reach further), and if the distance is larger then the elbow closes. On the other hand if the direction of the arm does not align with the target, the shoulder motor then applies an angular velocity to rotate the arm toward the target.

D. Evaluation details

D.1. Metrics

In the imitation learning experiments we report the following four evaluation metrics:

- **Boundaries**: We measure the accuracy of predicted boundary position. For each boundary latent variable b_i, we check if it exactly matches the ground truth task boundary, i.e., the point where a task ends and a new task begins. Let b_i denote the ground truth position for the i-th boundary, then the accuracy is defined as

$$\frac{1}{M-1}\sum_{i=1}^{M-1}\mathbb{I}[\arg\max_{b_i} q_{\phi_b}(b_i|x) = b_i],$$

  where $\mathbb{I}[x=y]$ denotes the Iverson bracket that returns 1 if $x = y$ and 0 otherwise.

- **Reconstruction**: This measures the average reconstruction accuracy of the original action sequence, given the ground truth state sequence, i.e., in a setting similar to teacher forcing:

$$\frac{1}{T}\sum_{i=1:M}\left(\sum_{j=b_{i'}:b_i-1}\mathbb{I}[\arg\max_{a_j}\pi_\theta(a_j|s_j,z_i) = a_j]\right),$$

  where $i' = i-1$ and $b_i = \arg\max_{b_i} q_{\phi_b}(b_i|x)$.

- **Exact match**: Here we measure the percentage of *exact matches* of full reconstructed action sequence (i.e., this score is 1 if all actions match for a single demonstration sequence and 0 otherwise), given the ground truth state sequence (provided one step at a time) as input.

- **Online eval**: Here, we first run our recognition model on a demonstration trajectory to obtain a sequence of latent codes. Then, we run the sub-task policy corresponding to the first latent code in the environment, until the termination policy predicts termination, in which case we move on to the next latent code, run the respective sub-task policy, and so on. We terminate if the episode ends (more than 200 steps, wrong object picked up or all tasks completed) and measure the obtained reward (either 0 or 1). For the baseline model, we infer a single latent code and run the respective policy until the end of the episode (without termination policy). We report the average reward obtained (multiplied by a factor of 100).

- **F1 Score**: To evaluate the pointer prediction performance for the continuous control task, we use the extra metric $F1$ score and optionally with a tolerance. In the continuous control setting, it is not easy to get the boundaries exactly correct as the transitions of the actions and observations across time steps are mostly smooth. The $F1$ score treats the predicted pointer locations and the ground truth pointer locations as 2 sets, and compute the precision as

$$\frac{\text{\#predictions that matches the ground truth}}{\text{total \#predictions}},$$

  irrespective of ordering, and recall as

$$\frac{\text{\#ground truth that has matches in predictions}}{\text{total \#ground truth}}.$$

  The $F1$ score is computed as

$$F1 = \frac{2\cdot\text{precision}\cdot\text{recall}}{\text{precision}+\text{recall}}.$$

A ‘match’ is considered to be successful if a predicted pointer location exactly equals a ground truth pointer location. With tolerance 1, a match is considered successful if the two are off by at most 1 time steps.

D.2. Segmentation baseline (LSTM surprisal)

To compare segmentation performance, we implemented a baseline algorithm based on auto-regressive behavioral cloning, termed *LSTM surprisal*. Given the state-action sequence $((s_1, a_1), (s_2, a_2), \dots, (s_T, a_T))$, this model maximizes the likelihood in the following form:

$$\max_\theta P_\theta(a_{1:T}|s_{1:T}) = \prod_{i=1}^{T} P(a_i|a_{1:i-1}, s_{1:i}) \tag{5}$$

Then, a natural approach to decide the segment boundary is based on the probability of each action. An action which is surprising (i.e., having low conditional probability) to the model should be an action that marks the beginning or end of a task segment.

Given the number of chunks M, we find the top $M-1$ boundary indicator variables $b_1, b_2, \dots, b_{M-1}$ with minimum conditional likelihood, i.e.,

$$\operatorname*{arg\,min}_{[b_1,b_2,\dots,b_{M-1}],b_i \le b_{i+1}} \sum_{i=1}^{M-1} P(a_{b_i}|a_{1:b_i-1}, s_{1:b_i}) \tag{6}$$

In the experiments, we use the same CNN (MLP for continuous control) architecture for encoding the state as in CompILE. An LSTM with same embedding size as our CompILE model is used here to model the dependency on the history of states and actions. We use the same training procedure as in the other models, i.e., we only train on 3x tasks, but report performance both on 5x. Interestingly, this model finds boundaries more consistently in the generalization setting (5 tasks) for the pick up task than in the setting it was trained on (3 tasks) in the grid world domain. We hypothesize that this is due to the fact that it has never seen a 4-th and 5-th object being picked up during training, and therefore assigns low probability to these events, which corresponds to a large “surprise” when these are observed in the generalization setting.

E. Qualitative results

Here, we provide qualitative analysis of the discovered subtask policies in the grid world environment. We run each sub-task policy for the pick up task on a random environment instance until termination, see Figures 3–4. The red cross marks the picked up object. We mark the policy in bold that the inference model of CompILE has inferred from a demonstration sequence for the task *pick up heart*.

In Figure 5, we investigate termination locations for the policies in the same trained CompILE model. We find that the model learns location-specific latent codes, which are effective at describing agent behavior from demonstrations. Nonetheless, the model can disambiguate close-by objects as can be seen in Figure 3.

References

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, 2018.

Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., Mohamed, S., and Lerchner, A. Beta-VAE: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2017.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with Gumbel-softmax. In *International Conference on Learning Representations*, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

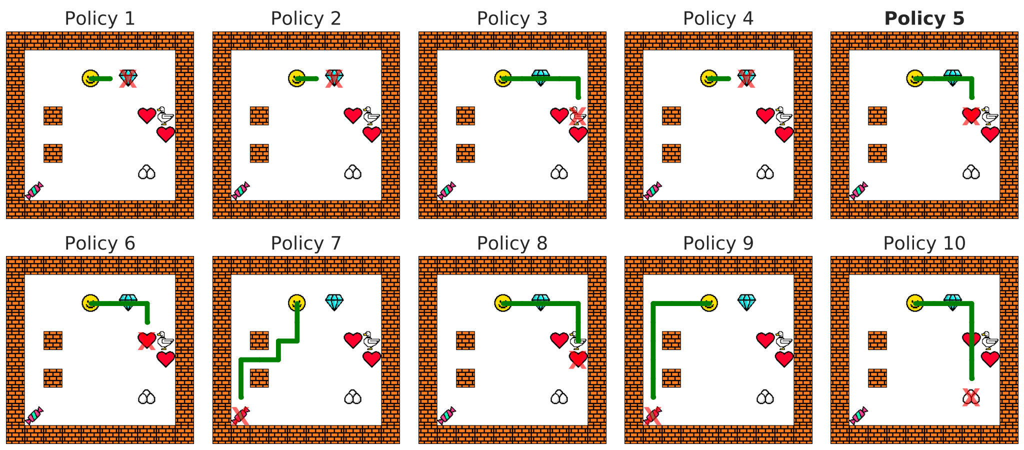

Figure 3: Example of sub-task policies discovered by the agent.

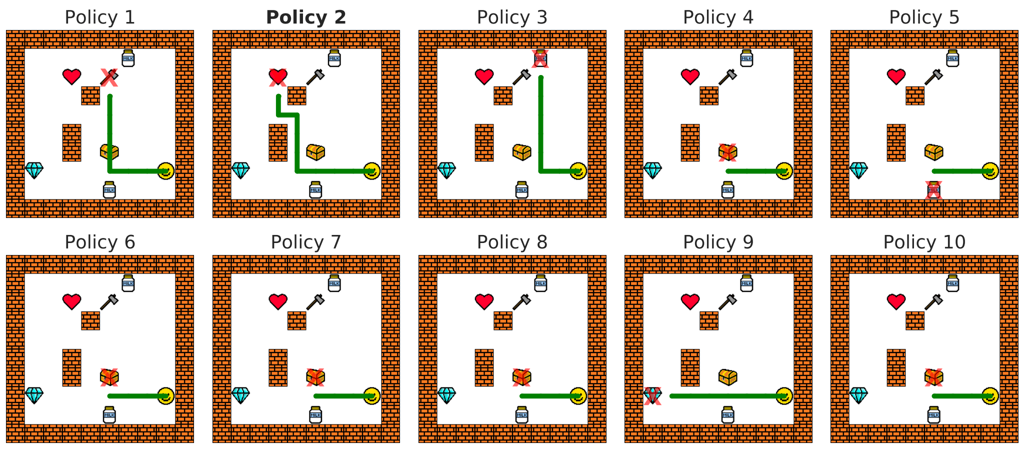

Figure 4: Example of sub-task policies discovered by the agent.

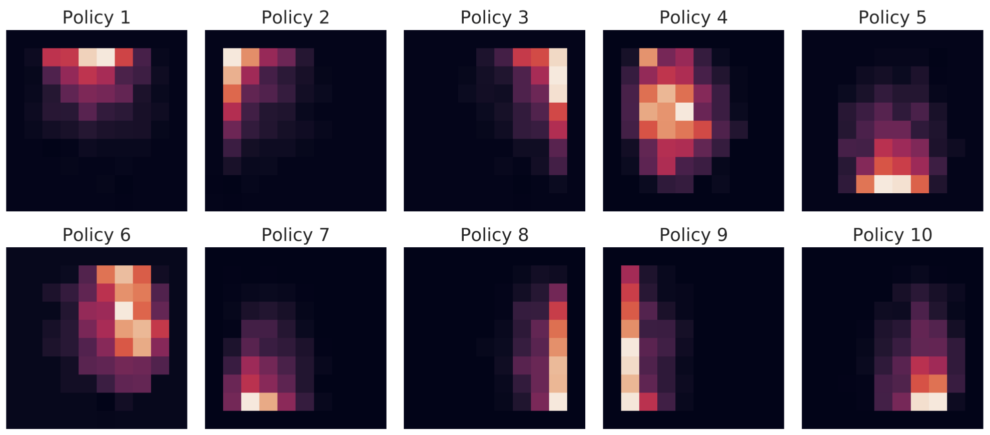

Figure 5: Heatmap of termination locations for each policy (for 1000 random environment instances).