Learning Object-Centric Representations for High-Level Planning in Minecraft

Steven James [1] **Benjamin Rosman** [1] **George Konidaris** [2]

Abstract

We propose a method for autonomously learning an object-centric representation of a high-dimensional environment that is suitable for planning. Such abstractions can be immediately transferred between tasks that share the same types of objects, resulting in agents that require fewer samples to learn a model of a new task. We demonstrate our approach on a series of Minecraft tasks to learn object-centric representations—directly from pixel data—that can be leveraged to quickly solve new tasks. The resulting learned representations enable the use of a task-level planner, resulting in an agent capable of forming complex, long-term plans.[1]

1. Introduction

Model-based methods are a promising approach to improving sample efficiency in reinforcement learning. However, they require the agent to either learn a highly detailed model—which is infeasible for sufficiently complex problems (Ho et al., 2019)—or to build a compact, high-level model that abstracts away unimportant details while retaining only the information required to plan. This raises the question of how best to build such an abstract model.

Recent work has shown how to learn an abstraction of a task that is provably suitable for planning with a given set of high-level actions (Konidaris et al., 2018). However, these representations are highly task-specific and must be relearned for any new task, or even any small change to an existing task. This makes them fatally impractical, especially for an agent that must solve multiple complex tasks.

We propose extending these methods by including additional structure—namely, that the world consists of objects, and that similar objects are common amongst tasks. This can substantially improve learning efficiency, because an object-centric model can be reused wherever that same object appears in a problem, and can also be generalised across objects that behave similarly—object *types*.

We assume that the agent is able to individuate objects in its environment, and propose a method for building object-centric abstractions given only the data collected by executing high-level skills. These abstractions specify both the abstract object attributes that support high-level planning, the object types, and an object-relative lifted transition model that can be instantiated in a new problem. This reduces the samples required to learn a new task by allowing the agent to avoid relearning the dynamics of previously-seen objects.

We demonstrate our approach in a series of Minecraft tasks (Johnson et al., 2016), where an agent autonomously learns PPDDL (Younes & Littman, 2004) representations of high-dimensional tasks from raw pixel input, and transfers these abstractions to new tasks, reducing the number of operators that must be learned from scratch.

2. Background

We assume that tasks are modelled as semi-Markov decision processes $\mathcal{M} = \langle \mathcal{S}, \mathcal{O}, \mathcal{T}, \mathcal{R} \rangle$ where (i) $\mathcal{S}$ is the state space, (ii) $\mathcal{O}(s)$ is the set of temporally-extended actions known as *options* available at state s, (iii) $\mathcal{T}$ describes the dynamics of the environment, specifying the probability of arriving in state s' after option o is executed from s, and (iv) $\mathcal{R}$ is the reward function. An option o is defined by the tuple $\langle I_o, \pi_o; \beta_o \rangle$, where I_o is the *initiation set* that specifies the states in which the option can be executed, π_o is the *option policy* which specifies the action to execute, and β_o specifies the probability of the option ceasing execution in each state (Sutton et al., 1999).

In this work, we use the object-centric formulation from Ugur & Piater (2015) to represent our state space $\mathcal{S}$: in a task with n objects, the state is represented by the set $\{\mathbf{f}_a, \mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n\}$, where $\mathbf{f}_a$ is a vector of the agent's features, and $\mathbf{f}_i$ is a vector of features or attributes particular to

[1]School of Computer Science and Applied Mathematics, University of the Witwatersrand, Johannesburg, South Africa [2]Department of Computer Science, Brown University, Providence RI 02912, USA. Correspondence to: Steven James <steven.james@wits.ac.za>.

Workshop on Object-Oriented Learning at ICML 2020.

[1]Results and videos can be found at the following URL: https://sites.google.com/view/mine-pddl

object i. Note that the feature vector describing each object can itself be arbitrarily complex, such as an image or voxel grid—in this work, we use pixels. Our state space representation differs from the standard approach in that individual objects have already been factored into their constituent low-level attributes. Practically, this means that the agent is aware that the world consists of objects, but is unaware of what the objects are, or if there are multiple instantiations of the same object present. It is also easy to see that different tasks will likely have differing numbers of objects with potentially arbitrary ordering; our abstract representation should thus be agnostic to this.

2.1. State Abstractions for Planning

We intend to learn an abstract representation suitable for planning. Prior work has shown that a sound abstract representation must necessarily be able to estimate the set of initiating and terminating states for each option (Konidaris et al., 2018). In classical planning, this corresponds to the *preconditions* and *effect* of each high-level action operator.

The precondition is defined as $\text{Pre}(o) = \Pr(s \in I_o)$, which is a probabilistic classifier that expresses the probability that option o can be executed at state s. Similarly, the effect or *image* represents the distribution of states an agent may find itself in after executing o from states drawn from some initial distribution Z: $\Pr(s' \mid s, o), s \sim Z$ (Konidaris et al., 2018). Since the precondition is a probabilistic classifier and the effect is a probabilistic density estimator, they can be learned directly from option execution data.

For large or continuous state spaces, estimating the effect of an option is impossible because the worst case requires learning a distribution conditioned on every state. However, if we assume that terminating states are independent of starting states, we can make the simplification $\Pr(s' \mid s, o) = \Pr(s' \mid o)$ (Konidaris et al., 2018). Such options are referred to as *subgoal* options (Precup, 2000).

Subgoal options are not overly restrictive, since they are options that drive an agent to some set of states with high reliability. However, it is likely an option may not be subgoal. In this case, we can *partition* an option's initiation set into a finite number of subsets, so that it is approximately subgoal when executed from any of the individual subsets. That is, we partition an option o's start states into finite regions $\mathcal{C}$ such that $\Pr(s' \mid s, o, c) \approx \Pr(s' \mid o, c), c \in \mathcal{C}$. As a result, the agent needs only to model $|\mathcal{C}|$ effects for each option.

3. Object-Oriented Abstract Representations

Although prior work (Konidaris et al., 2018) allows an agent to autonomously learn an abstract representation that enables fast task-level planning, generalisability is restricted—since the symbols are distributions over states in the current task, they cannot be reused in new ones. We now introduce an object-centric generalisation of a learned symbolic representation that admits transfer. Transfer here can be achieved when the state space representation contains features centred on objects in the environment. To achieve this, we must discover the types of objects, the abstract attributes that characterise each type, and the high-level operators describing the dynamics of the world.

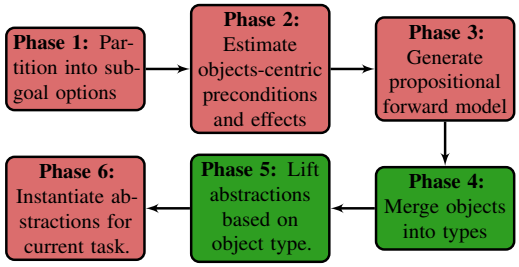


Figure 1: The full process of learning lifted representations. Red nodes represent problem-specific representations, while green nodes can be transferred between tasks.

3.1. Learning portable object-centric representations

An overview of our approach is given by Figure 1. We explain our approach by walking through the procedure on a single Minecraft task, and then demonstrate how these high-level representations can be transferred to new tasks.[2]

Our Minecraft task consists of five rooms with various items positioned throughout. Rooms are connected with either regular doors which can be opened by direct interaction, or puzzle doors which require the agent to pull a lever next to the door to open. The world is described by the state of each of the objects (given by each object's appearance as a 600×800 RGB image), the agent's view and inventory.[3]

The agent possesses high-level skills, such as `ToggleDoor` and `WalkToItem`. Execution is stochastic—opening doors occasionally fails, and the navigation skills are noisy in their execution. To solve the task, an agent must first collect the pickaxe, use it to break the gold and redstone blocks and collect the resulting items. It must then navigate to the crafting table, where it uses the collected items to first craft gold ingots and subsequently a clock. Finally, it must navigate to the chest and open it to complete the task. This requires an extremely long-horizon, hierarchical plan—the shortest plan that solves the task consists of 28 options that require *hundreds* of low-level continuous actions.

[2]Owing to space constraints, we defer the exact implementation and domain details to the appendix.

[3]To make learning easier, we compress the state space by downscaling images and applying PCA to a greyscaled version, preserving the top 40 principal components.

Phases 1–3 (as in Konidaris et al., 2018): After collecting transition data by executing options in the domain, the agent first partitions these options so that they approximately preserve the subgoal property. It then learns the preconditions for each option by fitting a classifier to each partition's initiation states. Next, a density estimator is used to estimate the effect of each partitioned option. The agent learns distributions over only the objects affected by the option, learning one estimator for each object. Each of these is a proposition in our PPDDL vocabulary $\mathcal{V}$.

For each partitioned option o, the agent has learned a precondition classifier $\hat{I}_o$ and effect estimator $\hat{\beta}_o$. In order to construct a PPDDL representation, however, the precondition and effects must be specified in terms of state distributions (propositions) only. Effects are modelled as such, and so pose no problem, but the learned precondition is a classifier rather than a state distribution. The agent must therefore determine which set of propositions from $\mathcal{V}$ best represents the precondition. This is achieved by replacing o's precondition with every $\mathcal{P} \in \wp(\mathcal{V})$[4] such that $\int_{\mathcal{S}} \hat{I}_o(s)\mathcal{G}(s)ds > 0, \mathcal{G} = \prod_{p\in\mathcal{P}} p$. In other words, the agent considers every combination of propositions $\mathcal{P}$ and samples data from their conjunction. If such data is classified as positive by $\hat{I}_o$, then $\mathcal{P}$ is used to represent the precondition. The preconditions and effects are now specified using distributions over state variables—this is our abstract propositional representation, suitable for planning.

Phases 4–5: At this point, the agent has learned an abstract, but task-specific, representation. Unfortunately, there is no opportunity for transfer, because each object is treated as unique. To overcome this, we must define the notion of object *types*.

In general, two objects are functionally identical if one object can be substituted for another while preserving all preconditions and effects. In practice, however, we can use a weaker condition to construct object types. Since the precondition for an object-centric skill usually depends only on the object it is interacting with, and because we have subgoal options that do not depend on the initial state, we can group objects by effects only.

Definition 1. Assume that option o has been partitioned into n subgoal options. Object i's *effect profile* under option o is denoted by $\text{EffectProfile}(i, o) = \{\mathcal{E}_i^{o(1)}, \ldots, \mathcal{E}_i^{o(n)}\}$, where $\mathcal{E}_i^{o(k)}$ is object i's effect distribution. Two objects i and j are *effect-equivalent* if $\text{EffectProfile}(i, o) = \text{EffectProfile}(j, o)$ for every o in $\mathcal{O}$.

Using the effects, the agent can determine whether objects i and j are similar, based on effect profiles, and if so, merge them into the same object class. Having determined the types, multiple propositions over objects of the same type are simply replaced with a single predicate parameterised by that class type. For example, if there are four doors in the domain, then the agent can replace four propositions representing each door closed with a single `ClosedDoor` predicate parameterised by an object of type `door`.

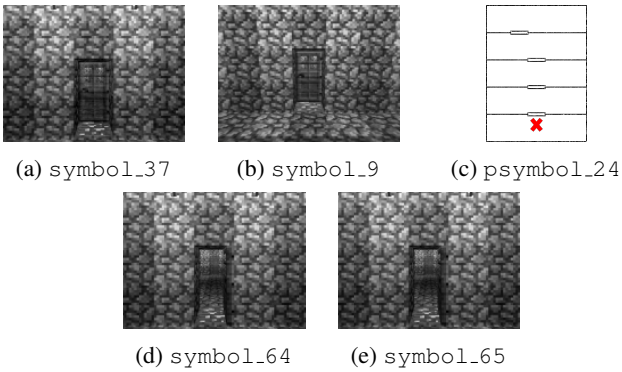
Phase 6: If the task dynamics are independent of the precise identity of each object, then our typed representation is sufficient for planning. However, in many domains the object-centric state space is *not* Markov. For example, in a task where only a particular key opens a particular door, the state of the objects alone is insufficient to describe dynamics—the identities of the key and door are necessary too. A common strategy in this case is to augment an ego- or object-centric state space with problem-specific, allocentric information to preserve the Markov property (Konidaris et al., 2012; James et al., 2018). We denote $\mathcal{X}$ as the space of problem-specific state variables, and $\mathcal{S}$ as the object-centric state space. The above complication does not negate the benefit of learning transferable abstract representations, as existing operators learned in $\mathcal{S}$ can be augmented with propositions over $\mathcal{X}$ on a per-task basis. In general, local information relative to individual objects will transfer between tasks, but problem-specific information, such as an object's global location, must be relearned each time.

For each partitioned option o with sets of start and end states $I_o, \beta_o \subseteq \mathcal{S} \times \mathcal{X}$, the agent re-partitions I_o such that $\Pr(x' \mid x_i, o) = \Pr(x' \mid x_j, o) \forall x_i, x_j \in \kappa, (\cdot, x') \in \beta_o$ for $\kappa \subseteq I_o$. This forms partitioned subgoal options in both $\mathcal{S}$ and $\mathcal{X}$. Denoting $\lambda \subseteq \mathcal{X}$ as the set of end states after re-partitioning, the agent can ground the operator by appending κ to the precondition and λ to the effect (if it differs from κ), where κ and λ are treated as problem-specific propositions. Finally, these problem-specific propositions must be linked with the grounded objects being acted upon. The agent therefore adds a precondition predicate conditioned on the identity of the grounded objects (see Figure 7 and the appendix for examples). Without this modification, it would be possible to open *any* door at that location. The final plan discovered by the agent is illustrated by Figure 3.

3.2. Transfer to Multiple Tasks

We next investigate transferring operators between five procedurally-generated tasks, where each task differs in the location of the objects and doors. The agent cannot thus simply use a plan found in one task to solve another. For a given task, we transfer all operators learned from previous tasks, and then continue to collect samples using uniform random exploration until we are able to produce a model which predicts that the optimal plan can be executed. We report the number of operators transferred between tasks averaged over 80 runs with random task orders (Figure 4).

[4] $\wp(\mathcal{V})$ denotes the powerset of $\mathcal{V}$.



(a) symbol_37 (b) symbol_9 (c) psymbol_24

(d) symbol_64 (e) symbol_65

```
(:action Toggle-Door-partition-1a
:parameters (?w - type0 ?x - type1)
:precondition (and (notfailed)
                   (symbol_37 ?w) (symbol_9 ?x)
                    (= (id ?x) 1) (psymbol_24))
:effect (and (symbol_64 ?x) (symbol_65 ?w)
             (not (symbol_9 ?x))
             (not (symbol_37 ?w)))
)
```

(f) A learned typed PDDL operator for one partition of the Toggle-Door option. The predicates underlined in red must be relearned for each new task, while the rest of the operator can be safely transferred.

Figure 2: Our approach learns that, in order to open a particular door, the agent must be standing in front of a closed door (symbol_37) at a particular location (psymbol_24), and the door must be closed (symbol_9). The effect of the skill is that the agent finds itself in front of an open door (symbol_64) and the door is open (symbol_65). type0 and type1 refer to the "agent" and "door" classes, while id is a fluent specifying the identity of the grounded door object, and is linked to the problem-specific symbol in red.

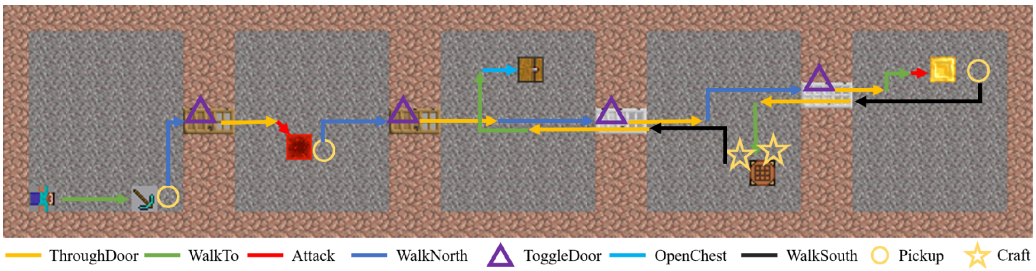


Figure 3: The path traced by the agent solving the first task. Coloured lines and shapes represent different option executions.

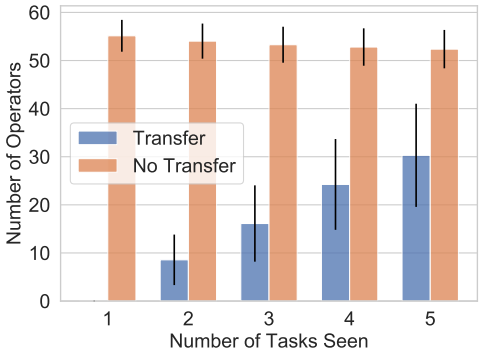


Figure 4: The orange bars represent the number of operators that must be learned to produce a sufficiently accurate model to solve the task. The blue bars represent the number of operators transferred between tasks. As the number of tasks increase, the number of new operators that must be learned decreases. Error bars represent one standard deviation.

4. Related Work and Conclusion

The most closely related work is that of Ugur & Piater (2015), who learn object-centric PDDL representations for planning. Their notion of an object type is similar, but object features are specified prior to learning, and discrete relations between object properties such as width and height are given. Object-oriented MDPs (Guestrin et al., 2003; Diuk et al., 2008; Marom & Rosman, 2018) specify states as sets of objects belonging to classes with associated attributes. We show how to learn an object-oriented representation along with the class types (which generalise over objects) as well as the abstract high-level dynamics model, while allowing for arbitrary effects.

Prior results in relational reinforcement learning have shown how to learn parameterised representations of skills (Finney et al., 2002; Pasula et al., 2004; Zettlemoyer et al., 2005), but the high-level symbols or attributes that constitute the state space are given. Conversely, we learn such representations from raw sensory input.

We have demonstrated how to learn a high-level, object-centric representation (including the type system, predicates and high-level operators) directly from pixel data. Our representation generalises across objects and can be transferred to new tasks, and provides an avenue for solving sparse-reward, long-term planning problems.

References

Ames, B., Thackston, A., and Konidaris, G. Learning symbolic representations for planning with parameterized skills. In *Proceedings of the 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018.

Andersen, G. and Konidaris, G. Active exploration for learning symbolic representations. In *Advances in Neural Information Processing Systems*, pp. 5016–5026, 2017.

Cortes, C. and Vapnik, V. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

Diuk, C., Cohen, A., and Littman, M. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 240–247. ACM, 2008.

Ester, M., Kriegel, H., Sander, J., and Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In *2nd International Conference on Knowledge Discovery and Data Mining*, volume 96, pp. 226–231, 1996.

Finney, S., Gardiol, N., Kaelbling, L., and Oates, T. The thing that we tried didn't work very well: deictic representation in reinforcement learning. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pp. 154–161, 2002.

Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N. Generalizing plans to new environments in relational MDPs. In *Proceedings of the 18th International Joint Conference on Artificial intelligence*, pp. 1003–1010. Morgan Kaufmann Publishers Inc., 2003.

Ho, M. K., Abel, D., Griffiths, T. L., and Littman, M. L. The value of abstraction. *Current Opinion in Behavioral Sciences*, 29:111–116, 2019.

James, S., Rosman, B., and Konidaris, G. Learning to plan with portable symbols. *ICML/IJCAI/AAMAS 2018 Workshop on Planning and Learning*, 2018.

Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. The malmo platform for artificial intelligence experimentation. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pp. 4246–4247, 2016.

Konidaris, G., Scheidwasser, I., and Barto, A. Transfer in reinforcement learning via shared features. *Journal of Machine Learning Research*, 13(May):1333–1371, 2012.

Konidaris, G., Kaelbling, L., and Lozano-Pérez, T. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61(January):215–289, 2018.

Marom, O. and Rosman, B. Zero-shot transfer with deictic object-oriented representation in reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 2297–2305, 2018.

Pasula, H., Zettlemoyer, L., and Kaelbling, L. Learning probabilistic relational planning rules. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pp. 73–81, 2004.

Platt, J. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

Precup, D. *Temporal abstraction in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 2000.

Rosenblatt, N. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, pp. 832–837, 1956.

Sutton, R., Precup, D., and Singh, S. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2): 181–211, 1999.

Ugur, E. and Piater, J. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *Proceedings of the 2015 IEEE International Conference on Robotics and Automation*, pp. 2627–2633, 2015.

Younes, H. and Littman, M. PPDDL 1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical report, 2004.

Zettlemoyer, L., Pasula, H., and Kaelbling, L. Learning planning rules in noisy stochastic worlds. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pp. 911–918, 2005.

A. Minecraft Task Description

Our Minecraft task consists of five rooms with various items positioned throughout. Rooms are connected with either regular doors, which can be opened by direct interaction, or puzzle doors, which require the agent to pull a lever next to the door to open. The world is described by the state of each of the objects (given by each object's appearance as a 600×800 RGB image), the agent's view, its location and current inventory. Figure 5 illustrates the state of each object in the world at the beginning of the task.

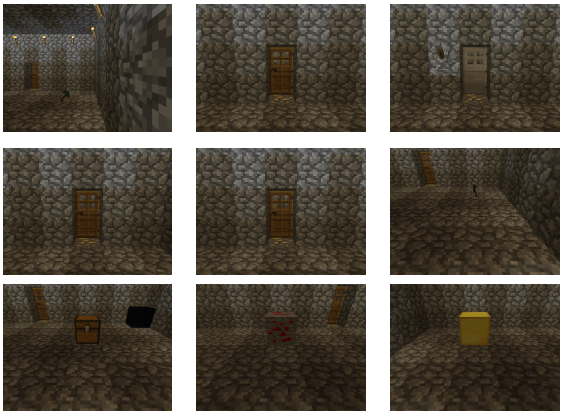


Figure 5: The state of each object in the world at the start of a task. From left to right, the images represent the agent's point of view, the four doors, the pickaxe, the chest, and the redstone and gold blocks. The inventory is not shown here.

The agent is provided with the following high-level skills: (i) `WalkToItem`—the agent will approach an item if it is in the same room. (ii) `AttackBlock`—the agent will break a block, provided it is near the block and holding the pickaxe. (iii) `PickupItem`—the agent will collect the item if it is standing in front of it. (iv) `WalkToNorthDoor`—the agent will approach the northern door in the current room. (v) `WalkToSouthDoor`—the agent will approach the southern door in the current room. (vi) `WalkThroughDoor`—the agent will walk through a door to the next room, provided the door is open. (vii) `CraftItem`—the agent will create a new item from ingredients in its inventory, provided it is near the crafting table. (viii) `OpenChest`—the agent will open the chest, provided it is standing in front of it and possesses the clock. (ix) `ToggleDoor`—the agent will open or close the door directly in front of it. The execution of these skills is stochastic—opening doors occasionally fails, as does attacking and collecting objects, and the navigation skills are noisy in their execution. Note that many of these options are either primitive actions or short chains of composed low-level actions.

The agent begins in the first room with an empty inventory, and must open the chest while holding a clock in order to complete the task. To achieve this, the agent must locate and collect the pickaxe. Then it must find both the gold and redstone blocks, break each using the pickaxe, and collect the resulting item. Having collected both the gold and redstone, it must navigate to the crafting table, where uses the collected items to first craft gold ingots, and subsequently a clock. Finally, it must navigate to the chest and open it to complete the task. This requires an extremely long-horizon, hierarchical plan—the shortest plan that solves the first task consists of 28 options that require *hundreds* of low-level continuous actions to be executed.

B. Learning a Portable Representation for Minecraft

In order to learn a high-level representation, we first apply a series preprocessing steps to reduce the dimensionality of the state space. We downscale images to 160×120 and then convert the resulting images to greyscale. We apply principal component analysis to a batch of images collected from the different tasks and keep the top 40 principal components. This allows us to represent each object (except the inventory, which is a one-hot encoded vector of length 5) as a vector of length 40.

Partitioning We collect data from a task by executing options uniformly at random. We record state transition data as well as, for each state, which options could be executed. We then partition options using the DBSCAN clustering algorithm (Ester et al., 1996) to cluster the terminating states of each option into separate effects. This approximately preserves the subgoal property, as described in Section 2 and previous work (Andersen & Konidaris, 2017; Konidaris et al., 2018; Ames et al., 2018).

Preconditions Next, the agent learns a precondition classifier for each of these approximately partitioned options using an SVM (Cortes & Vapnik, 1995) with Platt scaling (Platt, 1999). We use states initially collected as negative examples, and data from the actual transitions as positive examples. We employ a simple feature selection procedure to determine which objects are relevant to the option's precondition. We first compute the accuracy of the SVM applied to the object the option operates on, performing a grid search to find the best hyperparameters for the SVM using 3-fold cross validation. Then, for every other object in the environment, we compute the SVM's accuracy when that object's features are added to the SVM. Any object that increases the SVM accuracy is kept. Having determined the relevant objects, we fit a probabilistic SVM to the relevant objects' data.

Effects A kernel density estimator (KDE) (Rosenblatt, 1956) with Gaussian kernel is used to estimate the effect of each partitioned option. We learn distributions over only the objects affected by the option, learning one KDE for each object. We use a grid search with 3-fold cross validation to find the best bandwidth hyperparameter for each estimator. Each of these KDEs is an abstract symbol in our propositional PDDL representation.

Propositional PDDL For each partitioned option, we now have a classifier and set of effect distributions (propositions). However, to generate the PDDL, the precondition must be specified in terms of these propositions. We use the same approach as Konidaris et al. (2018) to generate the PDDL: for all combinations of valid effect distributions, we test whether data sampled from their conjunction is evaluated positively by our classifiers. If they are, then that combination of distributions serves as the precondition of the high-level operator.

Type Inference Before determining the class each object belongs to, we first define the notion of an object type:

Definition 2. Assume that option o has been partitioned into n subgoal options $o(1), \ldots, o(n)$. Object i's *profile* under option o is denoted by

$$\text{Profile}(i, o) = \left\{\{\mathcal{P}_i^{o(1)}, \mathcal{E}_i^{o(1)}\}, \ldots, \{\mathcal{P}_i^{o(n)}, \mathcal{E}_i^{o(n)}\}\right\},$$

where $\mathcal{P}_i^{o(k)}$ is the distribution over object i's states present in the precondition for partition k, and $\mathcal{E}_i^{o(k)}$ is object i's effect distribution.[5]

Definition 3. Two objects i and j are *option-equivalent* if, for a given option o, $\text{Profile}(i, o) = \text{Profile}(j, o)$. Furthermore, two objects are *equivalent* if they are option-equivalent for every o in $\mathcal{O}$.

The above definition implies that objects are equivalent if one object can be substituted for another while preserving the abstract preconditions and effects. Such objects can be grouped into the same object type, since for the purpose of planning, they are functionally indistinguishable. In practice, however, we can use a weaker condition to construct object types. Since the precondition for an object-centric skill usually depends only on the object it is interacting with, and because we have subgoal options that do not depend on the initial state, we can group objects by effects only.

Definition 4. Assume that option o has been partitioned into n subgoal options. Object i's *effect profile* under option o is denoted by

$$\text{EffectProfile}(i, o) = \left\{\mathcal{E}_i^{o(1)}, \ldots, \mathcal{E}_i^{o(n)}\right\},$$

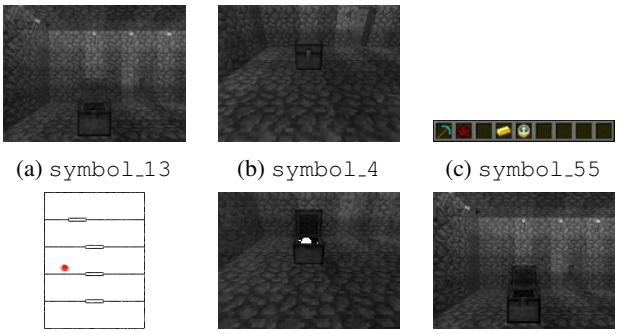
where $\mathcal{E}_i^{o(k)}$ is object i's effect distribution. Two objects i and j are *effect-equivalent* if $\text{EffectProfile}(i, o) = \text{EffectProfile}(j, o)$ for every o in $\mathcal{O}$.

To determine the type of each object, we assume first that they all belong to their own class. For each pair of objects i and j, we determine whether effect profiles are similar. This task is made easier because certain objects do not undergo effects with certain options. For example, the gold block cannot be toggled, while a door can. Thus it is easy to see that they are not of the same type. To determine whether two distributions are similar, we simply check whether the KL-divergence is less than a certain threshold. Having determined the types, we can simply replace all similar propositions with a predicate parameterised by an object of that class type.

C. Visualising Operators for Minecraft

In Figures 6–9, we illustrate some learned operators for the Minecraft tasks. All predicates and operators can be found at the following URL: `https://sites.google.com/view/mine-pddl`.

[5] These precondition and effect distributions can be null if the object is not acted upon.



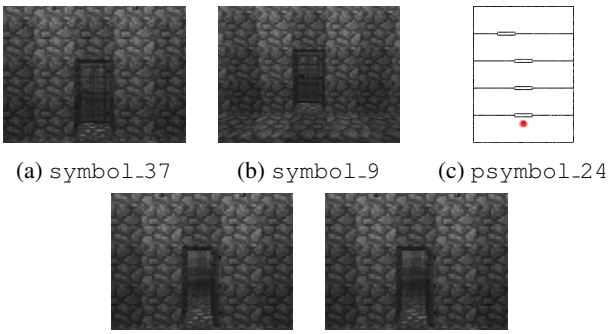
(a) symbol_13 (b) symbol_4 (c) symbol_55

(d) psymbol_8 (e) symbol_58 (f) symbol_59

```
(:action Open-Chest-partition-0
 :parameters (?w - type0 ?x - type6 ?y - type9)
 :precondition (and (notfailed) (symbol_13 ?w)
                    (symbol_4 ?x) (symbol_55 ?y)
                    (psymbol_8))
 :effect (and (symbol_58 ?x) (symbol_59 ?w)
              (not (symbol_4 ?x)) (not (symbol_13 ?w)))
)
```

(g) A learned typed PDDL operator for the Open-Chest skill. The predicate in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

Figure 6: Our approach learns that, in order to open a chest, the agent must be standing in front of a chest (symbol_13), the chest must be closed (symbol_4), the inventory must contain a clock (symbol_55) and the agent must be standing at a certain location (psymbol_8). The result is that the agent finds itself in front of an open chest (symbol_58) and the chest is open (symbol_59). type0 refers to the "agent" class, type6 the "chest" class and type9 the "inventory" class.



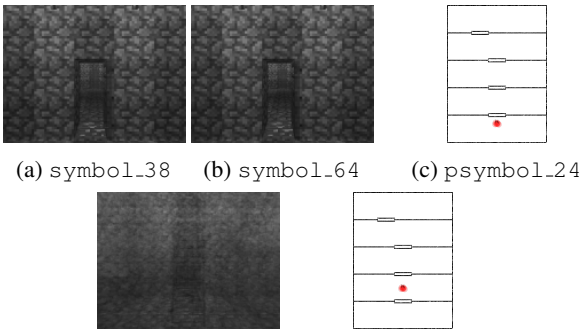
(a) symbol_37 (b) symbol_9 (c) psymbol_24

(d) symbol_64 (e) symbol_65

```
(:action Toggle-Door-partition-1a
 :parameters (?w - type0 ?x - type1a)
 :precondition (and (notfailed) (symbol_37 ?w)
                    (symbol_9 ?x) (psymbol_24))
 :effect (and (symbol_64 ?x) (symbol_65 ?w)
              (not (symbol_9 ?x)) (not (symbol_37 ?w)))
)
```

(f) A learned typed PDDL operator for a partition of the Toggle-Door option. The predicate in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

Figure 7: Our approach learns that, in order to open a particular door, the agent must be standing in front of a closed door (symbol_37) at a particular location (psymbol_24), and the door must be closed (symbol_9). The effect of the skill is that the agent finds itself in front of an open door (symbol_64) and the door is open (symbol_65). type0 refers to the "agent" class, type1a refers to an instantiation of a "door" class, which is a subclass of type1.



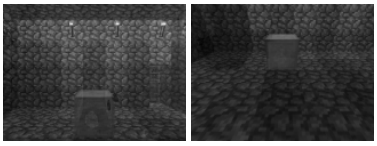
(a) symbol_38 (b) symbol_64 (c) psymbol_24

(d) symbol_50 (e) psymbol_12

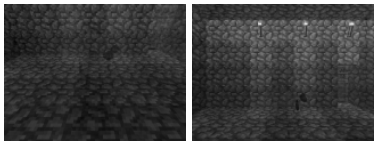
```
(:action Through-Door-partition-3-256a
 :parameters (?w - type0 ?x - type1a)
 :precondition (and (notfailed) (symbol_38 ?w)
                    (symbol_64 ?x) (psymbol_24))
 :effect (and (symbol_50 ?w) (psymbol_12)
              (not (symbol_38 ?w)) (not (psymbol_24)))
)
```

(f) Typed PDDL operator for a partition of the Through-Door option. The predicate in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

Figure 8: Abstract operator that models the agent walking through a door. In order to do so, the agent must be standing in front of an open door (symbol_38) at a particular location (psymbol_24), and the door must be open (symbol_64). As a result, the agent finds itself in the middle of a room (symbol_50) at a particular location (psymbol_12).



(a) symbol_15 (b) symbol_2 (c) psymbol_17



(d) symbol_19 (e) symbol_20

```
(:action Attack-partition-0-76a
 :parameters (?w - type0 ?x - type7)
 :precondition (and (notfailed) (symbol_15 ?w)
                    (symbol_2 ?x) (psymbol_17))
 :effect (and (symbol_19 ?x) (symbol_20 ?w)
              (not (symbol_2 ?x)) (not (symbol_15 ?w)))
)
```

(f) Typed PDDL operator for a partition of the Attack option. The predicate in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

Figure 9: Abstract operator that models the agent attacking an object. In order to do so, the agent must be standing in front of a gold block (symbol_15) at a particular location (psymbol_17), and the gold block must be whole (symbol_2). As a result, the agent finds itself in front of a disintegrated block (symbol_20), and the gold block is disintegrated (symbol_19).