QORA: Zero-Shot Transfer via Interpretable Object-Relational Model Learning

Gabriel Stella [1] **Dmitri Loguinov** [1]

Abstract

Although neural networks have demonstrated significant success in various reinforcement-learning tasks, even the highest-performing deep models often fail to generalize. As an alternative, object-oriented approaches offer a promising path towards better efficiency and generalization; however, they typically address narrow problem classes and require extensive domain knowledge. To overcome these limitations, we introduce *QORA*, an algorithm that constructs models expressive enough to solve a variety of domains, including those with stochastic transition functions, directly from a domain-agnostic object-based state representation. We also provide a novel benchmark suite to evaluate learners' generalization capabilities. In our test domains, QORA achieves 100% predictive accuracy using almost four orders of magnitude fewer observations than a neural-network baseline, demonstrates zero-shot transfer to modified environments, and adapts rapidly when applied to tasks involving previously unseen object interactions. Finally, we give examples of QORA's learned rules, showing them to be easily interpretable.

1. Introduction

Reinforcement learning, one of the primary branches of machine learning, encompasses problems where an agent, situated in some environment, makes decisions to maximize a reward signal (Kaelbling et al., 1996; Glanois et al., 2021). This makes the field uniquely applicable to real-world tasks such as robotic manipulation (Nagabandi et al., 2020), autonomous driving (Kiran et al., 2022), and plasma confinement for nuclear fusion (Degrave et al., 2022). There is hope that the study of reinforcement learning may even lead to human-level general intelligence, but current methods are far from achieving this goal in at least three aspects.

First, humans *generalize* well, transferring knowledge quickly to new settings, which allows learning to compound over time. Machine-learning methods, on the other hand, often fail when tested in scenarios that differ only slightly from how they were trained (Kansky et al., 2017). Second, current algorithms do not produce models that are *interpretable* to humans (Ghorbani et al., 2019; Glanois et al., 2021). Exchanging information is one of the most important features of social interaction, increasing trust and making knowledge acquisition more efficient; models that are difficult to understand cannot help with either of these goals. Third, humans learn *efficiently*, with many being able to confidently accomplish new tasks within hours or even minutes, while state-of-the-art reinforcement-learning agents can require decades worth of training experience (Badia et al., 2020). We refer to the set of these three properties as *GIE* (generalization, interpretability, efficiency) and propose the pursuit of all three as a path towards developing human-level intelligent systems. As a first step in this direction, we study algorithms that learn to understand their environments through interaction and observation.

More formally, the task we are interested in is *object-oriented transition modeling without domain knowledge*, which we describe here. Object-oriented representations work with entities and their attributes (e.g., position, color) rather than the tensors commonly used in deep learning (Diuk et al., 2008). Investigations in both machine learning (Chang et al., 2016) and human psychology (Spelke, 1990) have shown that understanding the world through objects and their relationships is an essential part of intelligence. Transition modeling is the process of learning predictive models of an environment's behavior; studies have shown that this is one of the primary features that enables humans to generalize (Hamrick et al., 2011) and that it can improve knowledge transfer in reinforcement learning as well (Young et al., 2022). Domain knowledge is additional information given to the agent beyond what it observes through its interactions, e.g., a list of domain-specific preconditions (Diuk et al., 2008; Marom & Rosman, 2018). While it is useful in certain applications, this information can be difficult or impossible to acquire; therefore, algorithms that operate without requiring such information are much more widely

[1]Department of Computer Science & Engineering, Texas A&M University, College Station, TX, USA. Correspondence to: Gabriel Stella <gabrielrstella@tamu.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

applicable. Although the combination of these constraints yields a challenging problem, investigating the topic is a principled path towards better techniques that possess GIE.

Unfortunately, domain-agnostic learning of object-oriented causal models is not well-studied; prior developments in reinforcement learning typically lack one or more of the three requirements stated in the preceding paragraph. Popular deep-learning approaches such as DQN (Mnih et al., 2015), Relational A2C (Zambaldi et al., 2019), Agent57 (Badia et al., 2020), and MuZero (Schrittwieser et al., 2020) do not explicitly interact with objects. The GNN architecture introduced in (Sancaktar et al., 2022) is object-oriented, but requires domain knowledge. Another line of research has studied non-deep-learning approaches to reinforcement learning, though these methods have their own limitations. Džeroski et al. (2001) introduced relational reinforcement learning, which uses first-order logical decision trees (Blockeel & De Raedt, 1998; Driessens et al., 2001) to represent policy and value functions, meaning that it does not model transitions. DOORMAX (Diuk et al., 2008) and its extensions (Hershkowitz et al., 2015; Marom & Rosman, 2018) tackle object-oriented transition modeling, but they require extensive domain knowledge to function. Schema networks (Kansky et al., 2017) use a domain-specific object representation. To our knowledge, the only notable existing techniques that satisfy all of the requirements of our problem are Multi-Head Dot Product Attention (MHDPA) modules (Vaswani et al., 2017) and the Neural Physics Engine (NPE) (Chang et al., 2016). These methods work with objects, can be used for transition modeling, and do not require domain knowledge; however, as we show later in this paper, they do not possess GIE, which calls for the investigation of alternative methods.

Towards this end, we introduce *QORA* (*Quantified Object Relation Aggregator*), a novel algorithm for learning object-oriented transition models without domain-specific knowledge. Unlike deep-learning approaches such as MHDPA and NPE, QORA extracts relational predicates from its observations and uses statistical methods to assemble these predicates into informative hypotheses. In order to determine how well QORA meets the goal of GIE, we evaluate it in several new object-oriented benchmark environments. Through these experiments, we show that QORA improves upon prior work in all three aspects. In particular, we demonstrate: a reduction in sample complexity of almost 10,000$\times$ relative to neural-network methods; zero-shot transfer to more complex scenarios; continual learning of new interactions; and easily-interpretable rules based on elegant first-order logic formulas. Thus, QORA lays a promising path towards progress on the challenging problem of achieving GIE, opening the door for future work to tackle tasks such as learning more-complex environments, playing difficult games, and controlling physical robots. The source code of both QORA's reference implementation and our benchmark suite are available online (Stella, 2024).

2. Object-Oriented Reinforcement Learning

Reinforcement learning involes an *agent* (e.g., a machine-learning algorithm) interacting with an *environment* (e.g., a game). In each step, a snapshot of the environment's state s is passed to the agent. The agent then chooses an action a, which is sent to the environment. The environment then *transitions* to a new state s'. This transition may also involve a scalar *reward signal* r. The agent is able to observe the results of its actions in the form of tuples (s, a, s', r). Typically, the agent's goal is to choose actions that maximize the accumulated reward signal (i.e., score) over time.

As part of the learning process, it is often helpful for the agent to utilize a model of the environment's transition dynamics. This model is trained to predict the future state that will result from the agent's action; in other words, the inputs to the model are (s, a) pairs and the outputs are s' states. This transition modeling is the problem we study in this paper. As such, we forgo the environment's reward signal, instead employing the *random agent* that chooses actions uniformly at random. The resulting observations are fed directly to the model learner.

Unlike many prior works, we operate in an explicitly object-oriented framework. Here, environments consist of a tuple (M, C, S, B, A, T). M is the set of *member attribute types*, each of which has a name (e.g., "position") and a size (the dimensionality of its values, e.g., two for (x, y) coordinates, three for RGB color). C is the set of *class types*, with each class $c \in C$ consisting of a name (e.g., "player") and a set of attribute types $c.attributes \subseteq M$. Every object o belongs to a class and contains attribute values corresponding to the attribute types of its class. Each attribute value is a d-element integer vector where d is the size of the attribute's type. S is the set of all valid states, where each state s is represented as a set of objects $\{o_i\}$. B is the distribution of initial states, which may be parameterized (by, e.g., width, height, number of objects, etc) to produce specific types of levels for testing. A is the finite set of actions available to the agent. $T(s'|s, a)$ is the environment's transition probability distribution, which gives the probability of moving to state s' after taking action a in state s. Each object is given a unique integer id, arbitrarily assigned for starting states and not modified by T, so that the learner can detect what changes have occurred during a transition. Environments must satisfy the Markov property (Sutton & Barto, 2018), i.e., $T(s'|s, a)$ does not depend on any past transitions or other hidden information.

To make this formalism concrete, we describe one of our benchmark environments, the `doors` domain (see Appendix B for other environments). In this domain, there are two attribute types: two-dimensional `position` (or `pos` for short), taking values in $\mathbb{Z}^2$, and a one-dimensional attribute we call `color`, taking values in $\{0, 1\}$. There are three classes: `player`, `wall`, and `door`. We use $s.c$ to refer to the set of all objects in state s belonging to class c, i.e., $\{o_i \in s \mid o_i.class = c\}$. For example, $s.walls$ is the set of all `wall` objects in state s. Objects of the `player` and `door` classes have `position` and `color` attributes; walls have `position`, but no `color`. For an object o_1 of type $o_1.class = c$, we use the notation $o_1[m]$ to refer to the value of the object's attribute m, e.g.: $o_1[color] = 0$, $o_1[pos] = (3, -2)$.

The initial state distribution B in the `doors` domain has three parameters: the width and height of the generated levels and the number of doors. An 8×8 example state with four doors is shown in Fig. 8c in the Appendix. There are six actions: movement in each direction (`left`, `right`, `up`, `down`), `stay`, and `change-color`. Walls and doors are not affected by any action, meaning that none of their attributes' values ever change. The `stay` action does not affect the player. Each movement action attempts to shift the player's position by one unit in the noted direction; however, if the destination position is blocked by a wall or door, the player will not move. Note that doors only block movement if they are *not* the same color as the player. In order to move through a door, the player's color can be toggled with the `change-color` action (so long as it is not currently occupying the same space as a door).

The dynamics of the `doors` domain can be expressed using a set of functions that independently predict future values of a particular attribute type. To make the functions more concise, it is also helpful to separate them based on the action and class they apply to. We call these functions *rules*. For example, when the `right` action is taken in state s, the following function gives the new `position` of a player o_1 in the resulting state s':

$$f_1(o_1, s) = \begin{cases} o_1[pos] & P_1(o_1) \lor P_2(o_1) \\ o_1[pos] + (1, 0) & \text{otherwise,} \end{cases} \tag{1}$$

where

$$P_1(o_1) \equiv \exists o_2 \in \text{s.walls}\colon\ o_2[pos] - o_1[pos] = (1, 0) \tag{2}$$

$$\begin{aligned} P_2(o_1) \equiv \exists o_2 \in \text{s.doors}\colon\ & o_2[pos] - o_1[pos] = (1, 0) \\ & \land \neg(o_2[color] - o_1[color] = 0), \end{aligned} \tag{3}$$

which means that the player moves if not blocked by a wall or a door of a different color.

For another example, the following gives the new `color` of a player o_1 when the `change-color` action is taken:

$$f_2(o_1, s) = \begin{cases} o_1[color] & P_3(o_1) \\ 1 - o_1[color] & \text{otherwise,} \end{cases} \tag{4}$$

where

$$P_3(o_1) \equiv \exists o_2 \in \text{s.doors} : o_2[pos] - o_1[pos] = (0, 0). \tag{5}$$

Many of the other rules of this domain, e.g., the effect of the `left` action on the `color` attribute, as well as any that determine attributes of wall or door objects, are identity.

When testing in a domain such as `doors`, we measure the accuracy of an algorithm's model $\hat{T}$ by calculating the Earth Mover's Distance (Werman et al., 1985; Rubner et al., 1998) (EMD) between the estimated and true distributions over future states. This allows us to consider the actual deviation between predicted states, rather than just the difference between the probability values as with other statistical distance metrics. EMD does so by incorporating a ground distance metric that gives the distance between states. For this, we use $d(s_1, s_2) = |s_2 - s_1|_1$, where $s_2 - s_1$ computes the difference in each object's attribute value from s_1 to s_2 (i.e., diff) and $|\cdot|_1$ is the sum of the absolute value of each attribute value difference (i.e., summed L1 norm over all object attribute values). Our goal is to model T as closely as possible, i.e., minimize the EMD on arbitrary transitions.

Learning an environment's transition dynamics given *only* (s, a, s') observations, with no additional domain-specific knowledge or assumptions, is a significant challenge. Since object sets have no fixed order or size, states in this formulation are not always representable as scalars or vectors as typically assumed in prior work (Strehl et al., 2007). This precludes the use of any methods that require fixed-sized inputs. Additionally, unlike in previous work (Diuk et al., 2008; Marom & Rosman, 2018), the learner here is not given direct access to the most important information; instead, anything it needs (e.g., relational predicates) must be generated from s. The goal is therefore not just to learn the behavior of the environment, but also to discern what information must be extracted in order to compute this behavior.

While it would be possible to memorize every seen observation and compute predictions on (s, a) without using the actual content of s, this would be problematic for several reasons. The space S and therefore also $S \times A$ are extremely large: even if limiting to small (8×8) levels in our doors domain, $|S \times A|$ is in the trillions. More importantly, this is not an efficient way of learning. An ideal algorithm would produce a model of T that *generalizes*, i.e., after being trained on only a small fraction of possible transitions, it generates accurate predictions even on unseen inputs.

3. QORA

We now describe QORA, a novel object-oriented model-learning algorithm that generates probability distributions over predicted states s' given the current state s and action a. QORA is not based upon popular deep-learning methods, instead incorporating a series of novel techniques to produce a distinct algorithm. Because of the resulting complexity, an in-depth description cannot be presented in the limited space of this paper. Thus, throughout this section, we maintain a high-level discussion; where relevant, we refer to pseudocode listings in Appendix A, which also contains more detailed information about the algorithm.

QORA learns rules with a structure similar to that shown in Eqs. (1) and (4). We call the output values of these piecewise functions *outcomes* or *effects* and the information that determines which case will be selected *conditions*. Rules as learned by QORA follow a specific structure. First, the outcomes are always of the form $o_1[m] \mathrel{+}= c_i$ for some case-specific constant vector c_i. Since this c_i is the only varying quantity between different cases' outcomes, rules can be modeled by simply keeping track of these *deltas*. When learning, deltas can be calculated by subtraction: for an object o_1 in s and its corresponding o'_1 in s', the observed effect is $o'_1[m] - o_1[m]$. Second, conditions are based on first-order logic formulas composing "primitive" predicates of two forms:

$$P_{m,v}(o_1)\colon o_1[m] = v, \tag{6}$$

which checks whether the value of attribute m in object o_1 is equal to v, and

$$P_{m,v}(o_1, o_2)\colon o_2[m] - o_1[m] = v, \tag{7}$$

which checks whether the *difference* in values of attribute m between objects o_1 and o_2 is equal to some value v. These predicates are extracted from state observations (and quantified appropriately) to produce conditions as shown in Alg. 3. To make the algorithm more general, other types of predicates could be added, but even these two alone allow QORA to learn many useful transition functions.

With the above in mind, a rule as learned by QORA is a function $r_{c,m,a}(o_1, s)$ that predicts the change in attribute m of objects of type c when action a is taken. To make a prediction, QORA iterates through all objects' attributes and applies the relevant rule, as shown in Alg. 5. A rule could be something like

$$r_{player,pos,right}(o_1, s) = \begin{cases} (0,0) & P_1(o_1) \lor P_2(o_1) \\ (1,0) & \text{otherwise,} \end{cases} \tag{8}$$

where P_1 and P_2 are the same as in Eqs. (2) and (3). These rules can also express behaviors like the effect of the `change-color` action, shown in Eq. (4), by incorporating the current value of attribute m:

$$r_{player,color,change-color}(o_1, s) = \tag{9}$$
$$\begin{cases} 0 & P_3(o_1) \\ 1 & o_1[color] = 0 \\ -1 & o_1[color] = 1 \end{cases}$$

where P_3 is the same as in Eq. (5).

Learning these rules from scratch is challenging. Notably, the most difficult part of this task is *determining what information must be incorporated into the conditions*. For example, in Eq. (8), the conditions make use of P_1 (checking for an adjacent wall) and P_2 (checking for an adjacent different-colored door). How can a learner determine that these pieces of information are important, and moreover, that they are *all* that is important? Even in a small game world, there is a great deal of information (e.g., several hundred unique predicates of the types in Eqs. (6) and (7) in an 8×8 world of the `doors` domain), and the set of formulas that can be constructed using these predicates is massive.

QORA must somehow determine which of these formulas is correct, but without domain-specific assumptions, we cannot *a priori* rule out any possibilities: all valid formulas, up to arbitrary size, must be considered. Fortunately, as a general principle, we know that in many environments, the "correct" rules depend on only a small amount of information. Thus, to make the search tractable, we apply Occam's Razor: we begin with simple conditions, measure their predictive power over a series of observations, and combine those that perform well to produce better conditions for another iteration of this process (Alg. 2). This allows us to generate formulas as large as a given environment requires while still tending to keep them as small as possible. In addition, we use a novel representation of first-order logic formulas that allows us to reduce the size of the search space by exploring multiple formulas simultaneously.

Combining predicates would typically lead to a massive increase in the size of the search space due to the existence of different types of quantifiers and connectives. With multiple quantified expressions containing several inner conditions combined with an arbitrary sequence of conjunctions, disjunctions, and negations, it becomes infeasible to enumerate even just the formulas generated by combining existing conditions. Rather than enumerating each of these formulas explicitly, we encode conditions using a novel representation that implicitly considers *all possible connectives and quantifiers* in a formula. Thus, for example, the formulas $(o_1[color] = 1)$ and $(\mathcal{Q} o_2 \in C_1 : o_2[color] = 0)$, where $\mathcal{Q}$ represents any possibly quantifier, combine to produce

$$(o_1[color] = 1) \odot (\mathcal{Q} o_2 \in C_1 : o_2[color] = 0)\,, \tag{10}$$

where $\odot$ represents any possible connective. Rather than evaluating to true or false, these conditions produce a bit string using the process shown in Alg. 4. This value encodes a large amount of information in a single evaluation; essentially, it is equivalent to evaluating every possible instantiation of the equation all at once and checking which possibilities were satisfied in the current state. This can be used to check for the truth value of simple formulas, such as

$$(o_1[color] = 1) \wedge (\forall o_2 \in C_1 : o_2[color] = 0) \quad (11)$$

based on the above condition, as well as more complex formulas; for example, a single evaluation of the condition

$$\mathcal{Q}o_1 \colon P(o_1) \odot Q(o_1) \quad (12)$$

can determine the truth value of the statement

$$(\forall o_1 \colon P(o_1)) \wedge (\exists o_1 \colon Q(o_1)). \quad (13)$$

In addition to reducing the size of QORA's search space exponentially, this representation enables extremely simple translation to formulas of the form shown in the previous rule listings (Eqs. (8) and (9)). Furthermore, the produced bit strings can be used as indices into a table, allowing QORA to directly model function outputs by counting observations.

To model a rule's output distribution, QORA records observations in a table of counters indexed by (condition bit string, outcome) pairs. This table allows QORA to calculate joint and conditional probability estimates for each outcome. Treating all rules as probabilistic in this way allows QORA to construct and evaluate partially-informed rules, essentially treating all observed randomness as being due to a lack of information. Then, when conditions are combined to produce a better-informed rule, observed outcomes will be less random. The construction process naturally terminates when no additional information improves the best-known hypothesis, i.e., all necessary information has been considered. In a deterministic environment, this corresponds to perfect predictive accuracy with no random outcomes.

In the `doors` domain, there are three important pieces of information to consider when predicting the player's movement: (1) is there a wall next to the player, (2) is there a door next to the player, and (3) is that door the same `color` as the player. It is possible to make predictions without using any information at all, but this will yield a very poor model. Considering just some of the information, e.g., solely the presence of an adjacent wall, makes the model much more powerful, but it will still fail in some scenarios (i.e., when there is a door next to the player). Checking for a door next to the player further improves the model, since it allows the model to be 100% certain that the player can move when there is *not* a wall or door, but if there *is* a door, the model can't know whether the player will be able to move without looking at the door's `color`. Thus, only the final model – which looks at all three pieces of information – will be perfectly accurate. Until this full condition is found by the algorithm, the environment appears to be stochastic.

For our iterative condition-combining process to work, we need a way to evaluate and rank candidate rules by their predictive power. To this end, we define the *score* of a rule to be its *average confidence in observations' true outcomes*, i.e., for a rule r with probability estimates $\hat{P}$, the rule's expected estimated probability $\hat{P}(y|x)$ for a randomly-sampled (condition bit string, outcome) pair (x, y). This is given by

$$\mathcal{S}(r) = \sum_{(x,y)} \hat{P}(y|x)\hat{P}(x, y). \quad (14)$$

This metric takes values in $[0, 1]$, with one meaning that the rule has perfect predictive accuracy. Incomplete rules will have scores less than one, with better-informed rules (i.e., incorporating more of the necessary information, thus enabling better prediction) getting higher scores. This allows us to rank candidate rules using their scores, helping QORA determine which conditions to combine as shown in Alg. 2.

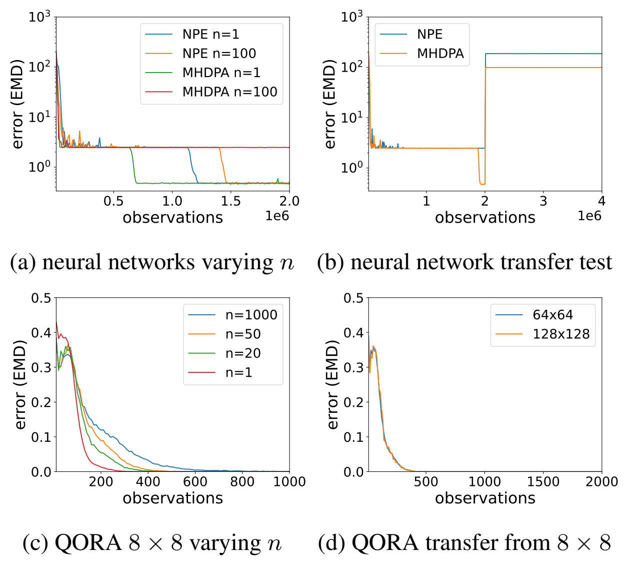
One final piece that makes QORA more robust is the use of confidence levels based on a hyperparameter $\alpha \in (0, 1)$. Rather than accepting conditions as soon as their $\mathcal{S}$ score exceeds the uninformed baseline, we estimate a confidence interval for each rule's $\mathcal{S}$ value and only accept once the lower bound of this interval exceeds the baseline. The confidence level is $1-\alpha$, so that as α is lowered towards zero, the confidence intervals become wider and QORA accepts candidates more conservatively. This significantly reduces the effect of spurious correlations, making QORA increasingly likely to incorporate only the most useful pieces of information into its rules. The choice of α is also somewhat domain-agnostic, as we show in Section 4.3; in more-challenging domains, the differences in $\mathcal{S}$ scores between candidates are smaller, so QORA automatically gathers more evidence before accepting hypotheses.

4. Experiments

To evaluate QORA and compare it to previous work, we set up tests that focus solely on each method's ability to learn from data. Each algorithm is run off-policy, with actions being chosen by a random agent and observations fed sequentially (i.e., online) *without* any sort of replay mechanism (Mnih et al., 2015). When running an experiment, we generate some number of initial states and step through n random actions in each to yield the total number of observations. The error metric we use is Earth Mover's Distance (EMD) as mentioned in Section 2. Our benchmark environments are described in Appendix B.

4.1. Comparison to Baselines

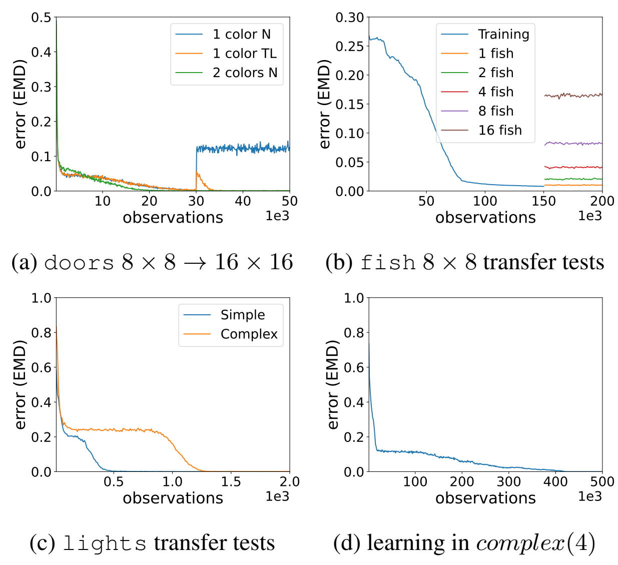
We begin by evaluating two neural-network architectures and QORA on the `walls` domain. Results are shown in Figure 1. QORA is run 100-1000 times per experiment to generate average results, but due to the computational cost, this could not be done for the neural networks; instead, after tuning hyperparameters and architectures, we report a single run with the best settings for each variant. The neural



(a) neural networks varying n (b) neural network transfer test

(c) QORA 8×8 varying n (d) QORA transfer from 8×8

Figure 1: Tests results from the walls domain.



(a) `doors` $8 \times 8 \to 16 \times 16$ (b) `fish` 8×8 transfer tests

(c) `lights` transfer tests (d) learning in $complex(4)$

Figure 2: QORA on advanced domains. (a) Testing continual learning with (TL) and without (N) additional training after transfer. (b) Transfer tests to larger worlds with more fish, showing that QORA scales optimally with the number of objects. (c) Pre-training in simpler scenarios to accelerate learning. (d) Tackling the challenging $complex(4)$ domain.

networks tested are two *object-based* architectures, which receive a list of objects similar to the representation QORA uses. NPE is an implementation of the Neural Physics Engine (Chang et al., 2016) *without* neighborhood masking to ensure it has no domain-specific information. MHDPA is an architecture using multi-head dot-product attention modules (Vaswani et al., 2017) to capture relations. Details on the architecture hyperparameters can be found in Appendix C.

Although both architectures are highly complex relative to the domain they are tested on, neither is able to achieve perfect accuracy even after two *million* observations, as shown in Figure 1a, where we train them in 8×8 worlds; they both saturate at around 0.5 error, which implies that they may have learned how walls work (i.e., that they do not move) but not how the player works (i.e., that it interacts with walls). Also note the effect of increasing n, the number of steps taken per level: the networks learn much slower, and in particular, the MHDPA with $n = 100$ never drops to 0.5 error. This is due to the reduced variety in the training data that comes with increasing n. In Figure 1b, we train the networks for two million observations in 8×8 worlds before transferring them without further training to 16×16 worlds. The substantial increase in error immediately after transfer indicates that the networks were relying on some extraneous details specific to the 8×8 levels.

In Figure 1c, we test QORA with varying values of n, showing again that lower n results in faster learning. Even in the worst case, though, QORA is still $3-4$ *orders of magnitude* more efficient than the deep-learning baselines. In Figure 1d, we see QORA demonstrate zero-shot transfer to a new type of level layout: after being trained for 1,000 observations on 8×8 levels, the learned model is transferred without further updates to 64×64 and 128×128 environments, the latter of which contain almost five *thousand* objects. The transferred models never make any errors; this shows that in all runs of the experiment, QORA deduced the correct relational rules (i.e., checking for adjacent walls) rather than using rules specific to the levels it was trained in. Specifically, QORA learns rules including the following:

$$r_{player,pos,right}(o_1, s) = \begin{cases} (0,0) & P(o_1) \\ (0,1) & \text{otherwise}, \end{cases} \tag{15}$$

where

$$P(o_1)\colon \exists o_2 \in \text{walls} : o_2[pos] - o_1[pos] = (0,1). \tag{16}$$

4.2. Tackling Challenging Domains

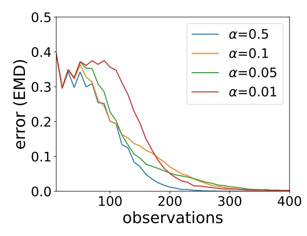
We now move to the more complex domains. Because the neural methods failed on the simpler `walls` domain, we only test QORA here. Beginning with `doors`, in Figure 2a we demonstrate several variations of $8 \times 8 \to 16 \times 16$ transfer after 30,000 observations: two tests beginning in a simpler variant of the domain (with only a single door color), with and without post-transfer model updates, and one test that trained in full-color worlds. Similarly to what we demonstrated in Figure 1d, when trained in 2-color worlds, QORA immediately transfers to larger levels with no additional error. In the runs where QORA trains in levels with only a single door color, QORA does not immediately achieve zero error post-transfer, but this is to be expected since it is impossible to distinguish between the "correct" hypothesis (i.e., comparing player color to door color) and other "incorrect" hypotheses (e.g., simply checking the player's current color). Still, QORA maintains its knowledge of walls, thereby achieving lower error post-transfer than an untrained model would. When post-transfer learning is enabled, QORA is able to quickly choose between the previously-equivalent hypotheses, re-converging to zero error within 5,000 additional observations.

We next evaluate QORA on the stochastic `fish` domain. Though the rules seem simple, this domain is extremely difficult to learn due to its random nature. The $\mathcal{S}$ score of the best candidate (i.e., the one that checks for walls around the fish) is only slightly higher than completely random guessing (≈ 0.3 vs. ≈ 0.2), making it difficult to determine which predicates are actually useful. Nonetheless, as shown in Figure 2b, QORA achieves low error within 100K observations, meaning that it is accurately predicting the entire *distribution* over possible future states.

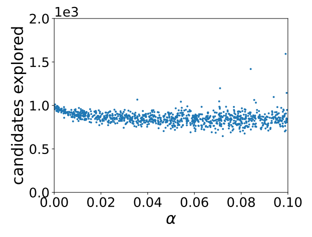
In the shown tests, after training for 150K observations in 8×8 worlds with a single fish, we transfer with no additional learning to larger levels with varying numbers of fish. When applied to levels with multiple fish, QORA is predicting the entire distribution of *every possible future position* of *all* of the fish. Because our error metric is cumulative, the error of a model that correctly handles each fish (independently from other fish, only looking at adjacent walls) will scale linearly with the number of fish, which is exactly what we see with QORA. The increased size of the levels post-transfer has no noticeable effect and the small amount of error that is present is simply due to the fact that QORA's probability estimates are based on observed frequencies.

We now move on to the `lights` domain. In Figure 2c, we compare two learning modes: "simple" (initial training for 1,000 observations in levels with 2-10 lights, then training on levels with 2-100 lights after transfer) and "complex" (training from scratch for 2,000 iterations on levels with 2-100 lights). The results demonstrate a major benefit of QORA's generalization capabilities: QORA can be trained rapidly in smaller, simpler scenarios and immediately transferred to more difficult settings. When compared to training QORA from scratch in more complex instances of the domain, we see that the pre-training allows QORA to converge to perfect accuracy almost $3\times$ faster.

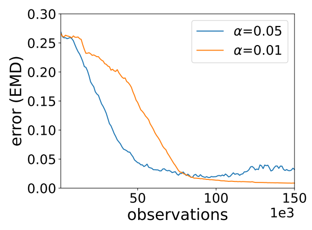
Finally, we apply QORA to the $\mathit{complex}(4)$ domain. This domain instance has five classes, two attributes, and twelve actions, in addition to the most elaborate rules (e.g., the behavior of the `jump` actions) among any of our domains. Still, QORA is able to reliably learn the domain perfectly.



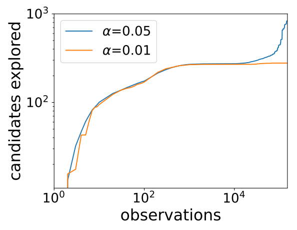
(a) `walls` error over time



(b) memory usage on `walls`



(c) `fish` error over time



(d) memory usage on `fish`

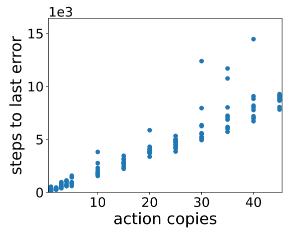
Figure 3: QORA performance varying α in 8×8 worlds.

In fact, as shown in Fig. 2d, QORA achieves zero error within $500K$ steps, which is *less than the time it took the neural-network baselines to reach 0.5 error on the* `walls` *domain*. Overall, the results in these challenging domains show that QORA possesses great efficiency and powerful knowledge-transfer capabilities.

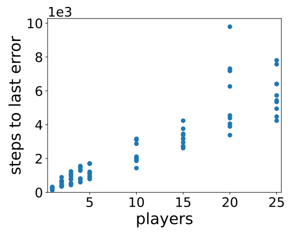
4.3. Varying α

In Figure 3, we test QORA with different values of its α hyperparameter in the `walls` and `fish` domains. This parameter controls QORA's willingness to accept uncertain hypotheses as "useful", leading to changes in its learning rate and memory useage. As seen in Figs. 3a and 3c, decreasing α (which corresponds to a higher confidence level and therefore larger intervals) leads to slower learning. Note, though, that QORA is quite robust to changes in this parameter; in the `walls` domain, nearly the entire range of possible α values is feasible to use. Additionally, simply choosing a value such as $\alpha = 0.01$ typically works and yields good performance across a wide variety of domains, from `walls` to `fish` as shown in Fig. 3c. In fact, we used $\alpha = 0.01$ in nearly all of the experiments in this paper.

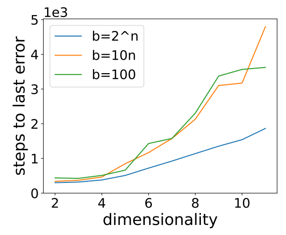
Although lowering α can lead to slightly slower learning, it also makes the learning process more stable by reducing the chance that QORA accepts somewhat-useful hypotheses due to spurious correlations. In Figure 3b, we plot the total number of candidates explored after 1,000 observations for many different runs with randomly-chosen α values in $[0, 0.1]$. Notice that lower α values lead to much more consistent memory usage. Conversely, using a higher α value increases the upper-bound on memory usage, leading



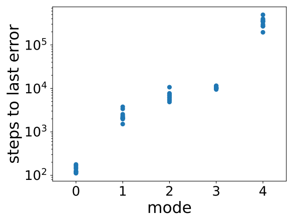
(a) `moves` domains



(b) `players` domains



(c) `paths` domains



(d) `complex` domains

Figure 4: The effect of environmental complexity on QORA's sample complexity. In (c), we vary the number of path objects b in relation to the dimensionality n.

QORA to occasionally generate a larger number of predicates than necessary. This occurs more often in challenging environments, especially stochastic ones, leading to the behavior shown in Fig. 3d where the use of a relatively high α value sometimes led QORA to incorporate unnecessary information into its conditions. It is important to note that the number of candidates shown in these figures is the sum total generated throughout training, *not* the number of candidates used for prediction: learned rules consist of only a single candidate per (c, m, a) triplet.

4.4. Scaling Environmental Complexity

To more thoroughly study the relationship between environmental complexity and QORA's sample complexity, we measured the number of observations taken by QORA to achieve 100% accuracy (i.e., the number of observations prior to its last error) in our four domain sets. In Figs. 4a and 4b, QORA's sample complexity appears to be approximately linear in the number of actions. This is due to the fact that the domains change in a uniform way as the domain parameter (i.e., number of players or actions) varies. Each additional stage simply adds new actions with rules similar to the prior ones. Because QORA learns rules independently for each (c, m, a) triplet, this just adds on some constant factor to the sample complexity.

However, this is not always the case. Fig. 4c shows similar experiments in the `paths` domains, but varying the way that the number of path objects b scales with the dimensionality of the environment n. In most cases, such as when the number of paths is constant or linear in n, this qualitatively changes the behavior of the environment from QORA's perspective. A notable exception is when the number of paths is exponential in the dimensionality. This choice ensures that the random walk process is essentially similar across values of n, thus leading to QORA's sample complexity being roughly linear in n.

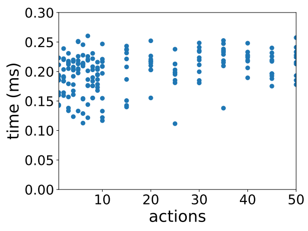
Finally, to show how QORA performs in extremely difficult environments, we evaluate its time to zero error in the `complex` domains. As shown in Fig. 4d, QORA is able to learn all of the environments perfectly. Notably, QORA's sample complexity increases nonlinearly with each additional step. Because each domain mode makes different types of changes to the transition function, the effective difficulty does not vary uniformly. The fact that QORA matches this trend is evidence that its performance is related in a meaningful way to the "true" complexity of the domain.

4.5. Runtime Complexity

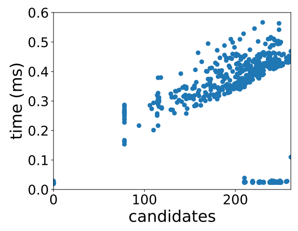
The time taken to compute QORA's prediction function is fairly low – about a tenth of a millisecond even in a large domain (`paths` with $n = 10$ and 200 `path` objects) and essentially constant over time. The observation function is more expensive, so that is what we will focus on here. Figs. 5a, 5c, and 5e show QORA's average per-step runtime vs. the number of actions in several of our domain sets. One can see that in some cases, there is a correlation between runtime and the number of actions, but this does not always happen. The apparent relationship in Figs. 5c and 5e is indirect, not causal. The runtime is actually based on the complexity of the environment, which happens to often relate in some way to the number of actions. Thus, in the `moves` domain (Fig. 5a), where each new copy of the movement actions has the same effect on environmental complexity, we see that the number of actions has no bearing on QORA's runtime.

As shown in Figs. 5b and 5d, QORA's runtime is determined mostly by the candidates it generates. By measuring the number of candidates that QORA has created at each step, we find a clear trend: more candidates leads to longer runtimes, until learning converges. This also causes QORA's observation runtime to increase as it learns, up until the point that it has converged, as shown in Fig. 5f. The increase in speed after convergence is due to QORA's boosting step, which skips most of the computation once the correct rules have been found. Hence, to produce a single estimate of QORA's runtime per run in Figs. 5a, 5c, and 5e, we averaged its per-step runtime prior to convergence.

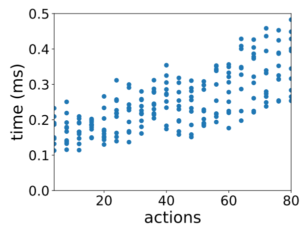
Predicting QORA's runtime in general is a significant challenge. It depends on the number of candidates QORA has generated, but this is again a difficult quantity to predict; it in turn depends on details of the environment such as the number of objects, the variety of observed states, the



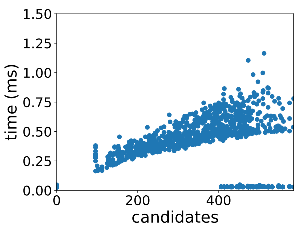
(a) runtime vs. number of actions in the `moves` domain



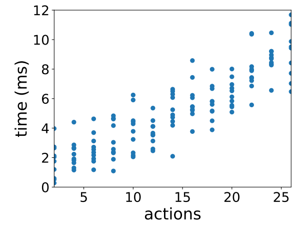
(b) runtime vs. number of candidates in the `moves` domain



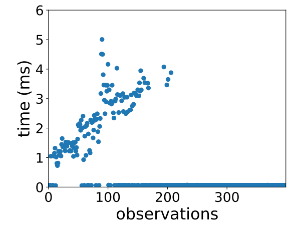
(c) runtime vs. number of actions in the `players` domain



(d) runtime vs. number of candidates in the `players` domain

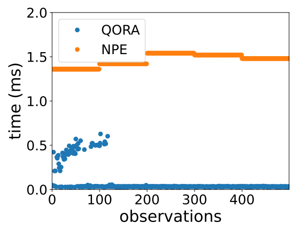


(e) runtime vs. number of actions in the `paths` domain

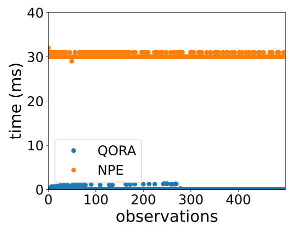


(f) runtime vs. number of observations in the 3D `paths` domain

Figure 5: QORA runtime results in various domain sets.



(a) walls 8×8



(b) 10×10 (no batching)

Figure 6: QORA and NPE runtime comparison in the `walls` domain.

complexity of the environment's rules, and the frequencies of observed events. In more challenging domains where QORA must combine many predicates together to form effective hypotheses, the candidates themselves become larger and more expensive to compute. Thus, although some trends are known, developing a full theoretical model of QORA's complexity (both sample and runtime) is an open problem.

Finally, we compare QORA's runtime to that of a standard neural-network implementation, PyTorch, on the same machine. Note that for fairness, PyTorch is run on the CPU. In Fig. 6a, we run both QORA and NPE in 8×8 levels from the `walls` domain. Here, NPE is using batching, so rates are averaged over each batch. Note that although QORA's runtime briefly increases, it is still several times faster than NPE. After convergence, QORA is approx. $50\times$ faster than NPE. In Fig. 6b, we disable batching on NPE (since QORA receives only a single sample at a time) and increase the size of the worlds to 10×10. Both QORA and NPE slow down compared to the first case, but NPE slows down so significantly that QORA's slowest step is still approx. $20\times$ faster than any NPE step. After convergence, QORA is approx. $600\times$ faster than NPE. Recall from Section 4.1 that QORA's sample complexity is also approx. $10,000\times$ lower than NPE's; combining these two factors yields a potential learning speedup of *six million times* over NPE.

4.6. Model-Based Planning

To assess QORA's applicability to solving tasks, we ran planning experiments in both the walls and doors domains. In each experiment, a random starting state was created and a goal state was generated by applying a random sequence of actions to the start state. We then used both QORA models and an "oracle" model (which uses the environment's true transition function T to make predictions) to create plans for the agent to move from the start state to the goal state. To produce plans using the models, we used Breadth First Search with the models' predictions serving as the transition function for the search. As expected, fully-trained QORA models perform identically to the oracle planner, finding optimal paths. Interestingly, incomplete QORA models (i.e., that did not perfectly model T) were often still able to produce successful plans, and because QORA learns so efficiently, even *completely untrained* models were able to quickly generate plans in the walls domain by taking observations while attempting to reach the goal state.

5. Conclusion

We introduced *QORA*, an algorithm capable of learning predictive models for a large class of domains by directly extracting information from object-oriented state observations. We demonstrated that QORA achieves zero-shot transfer using interpretable relational rules and is capable of rapid continual learning while simultaneously having orders-of-magnitude better sample efficiency than deep-learning approaches. This contribution opens a new path for future developments in transition modeling and the wider field of reinforcement learning.

Impact Statement

This paper explores a novel path towards addressing three notable limitations of popular deep-learning approaches: ineffective generalization, lack of interpretability, and poor data efficiency. Improving upon these qualities, which we refer to as GIE, will help future machine-learning algorithms have a more positive impact on the world. In particular: better generalization enables more powerful learning and problem-solving capabilities, allowing us to use machine learning to tackle more complex tasks; better interpretability leads to predictability, safety, and robustness, making maching-learning systems more trustworthy; and better efficiency helps reduce energy use, save time, and make machine learning more accessible to all. It is our hope that by focusing on these aspects "from the ground up", this work will set a new path for the development of technologies that can be safely applied to difficult problems.

References

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., and Blundell, C. Agent57: Outperforming the Atari human benchmark. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 507–517, Jul. 2020.

Blockeel, H. and De Raedt, L. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101 (1):285–297, May 1998.

Chang, M. B., Ullman, T., Torralba, A., and Tenenbaum, J. B. A compositional object-based approach to learning physical dynamics. *CoRR*, abs/1612.00341, 2016.

Degrave, J., Felici, F., Buchli, J., Neunert, M., Tracey, B., Carpanese, F., Ewalds, T., Hafner, R., Abdolmaleki, A., de Las Casas, D., et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

Diuk, C., Cohen, A., and Littman, M. L. An object-oriented representation for efficient reinforcement learning. In *Proc. ICML*, pp. 240–247, July 2008.

Driessens, K., Ramon, J., and Blockeel, H. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *Proceedings of the 12th European Conference on Machine Learning*, pp. 97–108, Sept. 2001.

Džeroski, S., De Raedt, L., and Driessens, K. Relational reinforcement learning. *Machine Learning*, 43:7–52, Apr. 2001.

Ghorbani, A., Abid, A., and Zou, J. Interpretation of neural networks is fragile. In *Proceedings of the AAAI conference on artificial intelligence*, pp. 3681–3688, 2019.

Glanois, C., Weng, P., Zimmer, M., Li, D., Yang, T., Hao, J., and Liu, W. A survey on interpretable reinforcement learning. *CoRR*, abs/2112.13112, Apr. 2021.

Hamrick, J., Battaglia, P., and Tenenbaum, J. B. Internal physics models guide probabilistic judgments about object dynamics. In *Proceedings of the 33rd annual conference of the cognitive science society*, pp. 1545–1550, Jul. 2011.

Hershkowitz, D. E., MacGlashan, J., and Tellex, S. Learning propositional functions for planning and reinforcement learning. In *2015 AAAI Fall Symposium Series*, pp. 38–45, Nov. 2015.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, Jan. 1996.

Kansky, K., Silver, T., Mély, D. A., Eldawy, M., Lázaro-Gredilla, M., Lou, X., Dorfman, N., Sidor, S., Phoenix, S., and George, D. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In *ICML*, pp. 1809–1818, 2017.

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A. A. A., Yogamani, S., and Pérez, P. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23 (6):4909–4926, Jun. 2022.

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, Dec. 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533, 2015.

Nagabandi, A., Konolige, K., Levine, S., and Kumar, V. Deep dynamics models for learning dexterous manipulation. In *Proceedings of the Conference on Robot Learning*, pp. 1101–1112, Oct. 2020.

Rubner, Y., Tomasi, C., and Guibas, L. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision*, pp. 59–66, Jan. 1998.

Sancaktar, C., Blaes, S., and Martius, G. Curious exploration via structured world models yields zero-shot object manipulation. In *Advances in Neural Information Processing Systems*, Nov. 2022.

Schapire, R. E. The strength of weak learnability. *Machine Learning*, 5(2):197–227, Jun. 1990.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839): 604–609, 2020.

Spelke, E. S. Principles of object perception. *Cognitive Science*, 14(1):29–56, Jan. 1990.

Stella, G. QORA. `https://github.com/GabrielRStella/QORA`, 2024.

Strehl, A. L., Diuk, C., and Littman, M. L. Efficient structure learning in factored-state mdps. In *AAAI*, pp. 645–650, Jul. 2007.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, Dec. 2017.

Werman, M., Peleg, S., and Rosenfeld, A. A distance metric for multidimensional histograms. *Computer Vision, Graphics, and Image Processing*, 32(3):328–336, Dec. 1985.

Young, K., Ramesh, A., Kirsch, L., and Schmidhuber, J. The benefits of model-based generalization in reinforcement learning. *arXiv preprint arXiv:2211.02222*, 2022.

Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., Shanahan, M., Langston, V., Pascanu, R., Botvinick, M., Vinyals, O., and Battaglia, P. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, May 2019.

A. QORA Pseudocode

QORA provides two high-level functions: `observe(s, a, s')`, which updates the model with a single training observation, and `predict(s, a)`, which applies the model. Here, s is the current state, a is the agent's chosen action, and s' is the resulting next state. As mentioned in Section 3.3 of the paper, QORA makes predictions independently for each (class, member attribute type, action) triplet; thus, it keeps a separate instance of the *Predictor* class for each such triplet. In the main QORA class' `observe` function, shown in Alg. 1, we loop through every member of every object, calculate the effect for that member (i.e., the change from s to s'), and use that information to update the corresponding Predictor. Predictor objects are stored in a map (i.e., dictionary) called `predictors` indexed by the (Class, Member, Action) triplet they correspond to.

Algorithm 1: QORA Observation Function

```
Func Qora.observe(State s, Action a, State s') → void
    for (Object obj in $s$) do
        Object obj' = $s'$.find(obj.id)    // find corresponding object data in next state
        for (Member $m$ in obj.attributes) do
            Value delta = obj'[$m$] - obj[$m$]    // calculate the change in attribute m's value, e.g., +(1, 0)
            Predictor p = predictors[obj.class, $m$, $a$]
            p.observe($s$, obj, delta)
```

To implement the Predictor class' `observe` function, shown in Alg. 2, we use three helper classes:

- Condition, which represents information that a potential rule may check to determine its output, e.g.,

$$f(o_1, s) : \mathcal{Q}o_2 \in \text{s.walls} : o_2[pos] - o_1[pos] = (-1, 0), \tag{17}$$

  where $\mathcal{Q}$ represents any possible quantifier;

- FrequencyTable, which implements the approximator table discussed in Section 3.1 of the paper and calculates the approximator's S score (Eq. 1 from the paper, stored as the `success` member);

- Candidate, which represents an entire rule, e.g.,

$$r(o_1, s) = \begin{cases} (0,0) & \exists o_2 \in \text{s.walls} : o_2[pos] - o_1[pos] = (-1, 0) \\ (-1,0) & \text{otherwise.} \end{cases} \tag{18}$$

  and stores a Condition (as its `condition` member) and a FrequencyTable (as its `table` member) to both learn the rule's outputs and estimate its predictive power. The S score of a Candidate's `table` corresponds to the utility of its `condition`: a useful Condition will give information that allows the table to maintain a high score.

The Predictor maintains several containers and a `baseline` estimator. The `observed` set records all previously-used Condition objects to ensure that no rule is added twice to the `working` list. The baseline learns the unconditional distribution of outputs that the Predictor has seen, i.e., it tracks how well the Predictor's observed data can be modeled using no information about the current state s. The `working` list contains all Candidate rules whose S scores are not higher than the baseline's score. These S scores are compared using confidence levels (QORA's α parameter), so as more data is collected, some Candidates' scores may exceed the baseline. When this occurs, the Candidate is moved to the `hypotheses` list.

The first step in `Predictor.observe` is to update the `baseline` guess and all Candidates in the `hypotheses` list, shown in Alg. 2 lines 10 – 13. This refines all of the corresponding distributions and S scores. After doing so, we check if there is a new best hypothesis. This is done in step two, shown in Alg. 2 lines 14 – 18. If the top hypothesis (i.e., with the highest S score) has changed, we construct new Candidates by combining the new top hypothesis with every other hypothesis. Specifically, we combine their Conditions, yielding Candidates that are more informed. This step demonstrates one of QORA's core concepts: if two well-performing Candidates use different information, a Candidate that looks at all of that information at once may be able to make better predictions than both. Since better predictions yield a higher S score, this more powerful Candidate will eventually be moved to the top, where it will be combined to again generate

Algorithm 2: Predictor Observation Function

```
   // Predictor class members:
1  set<Condition> observed
2  list<Candidate> working
3  FrequencyTable baseline
4  list<Candidate> hypotheses    // candidates that are better than the baseline

5  Func Predictor.checkAdd(Condition c) → void
6      if not (observed.contains(c)) then
7          observed.add(c)
8          working.add(Candidate(c, FrequencyTable()))    // add Candidate initialized with blank table
9  Func Predictor.observe(State $s$, Object target, Value effect) → void
       // 1. update the baseline and the current hypotheses
10     baseline.observe(0, effect)    // the baseline gets no information, so it always receives input 0
11     for (Candidate h in hypotheses) do
12         BitString c = h.condition.evaluate($s$, target)
13         h.table.observe(c, effect)
       // 2. check if there is a new best hypothesis
14     bubble_up(hypotheses)    // move the hypothesis with the highest S score (Eq. 1) to the front
15     if (best hypothesis has changed) then
16         Condition c1 = hypotheses[0].condition    // new best hypothesis' formula
17         for (Candidate h2 in hypotheses) do
18             checkAdd(c1 + h2.condition)    // combine formulas to produce new candidates
       // 3. boosting: ignore observations that we can already predict
19     if (current observation is expected by current best hypothesis) then
20         return
       // 4. extract all basic predicates from the observed state
       // e.g. "x[color] = (1)" or "y[position] - x[position] = (1, 0)"
21     for (Condition c in ExtractPredicates(s, target)) do
22         checkAdd(c)
       // 5. update candidates in the working set and check if any should be upgraded
23     for (Candidate w in working) do
24         BitString c = w.condition.evaluate(s, target)
25         w.table.observe(c, effect)
           // compare S scores (Eq. 1) using alpha confidence level
26         if (w.table.success > baseline.success) then
27             move w to hypotheses
28             checkAdd(w.condition + hypotheses[0].condition)
```

more-informed Candidates. This process continues until the best possible Candidate is found; for deterministic environments, this corresponds to a Candidate with $S = 1$ (perfect predictive power). For example, in the `doors` domain, at least three Conditions need to be combined: (1) checking for an adjacent wall, (2) checking for an adjacent door, and (3) checking that the adjacent door has the same color as the player. Each intermediate step (e.g., checking for an adjacent wall and an adjacent door, but ignoring the door's color) will have an S score higher than the baseline's but less than 1.

In step 3, lines 19 – 20 of Alg. 2, we perform boosting (Schapire, 1990) by discarding observations that are predicted by the current best hypothesis. This step provides two benefits: first, it skips the rest of the function, massively decreasing the amount of computation to be performed; second, it implicitly makes all Candidates in the `working` list conditional on the current best hypothesis, increasing the power of the information they receive and making them better suited to correct errors in that best hypothesis.

In step 4, lines 21 – 22 of Alg. 2, we add Candidates based on any newly-seen primivite predicates to the `working` list. These primitives form the building blocks of the powerful rules that QORA constructs. Alg. 3 shows our ExtractPredicates function. Note that in future work, this could easily be extended with additional predicate types.

The basic Candidates, as well as any more-complex rules still in the `working` list, are updated in step 5, lines 23 – 28 of

Algorithm 3: Predicate extraction function

```
Func ExtractPredicates(State $s$, Object target) → set<Condition>
    set<Condition> conditions
    // Non-quantified (target-only) predicates
    // For example, "P(x): x[color] = (1)"
    for (Member m in target.attributes) do
        // create a predicate of the form $P(x) : x[m] = v$
        Predicate p = CreateValuePredicate(m, target[m])
        // create a condition of the form $f(x) = P(x)$
        Condition c = CreateCondition(p)
        conditions.add(c)
    // Predicates involving objects other than the target
    // For example, "P(x, y): y[position] - x[position] = (1, 0)"
    for (Object other in $s$) do
        // Add pairwise predicates
        for (Member m in target.attributes) do
            // Verify that both objects possess this attribute
            if (other.attributes.contains(m)) then
                // create a predicate of the form $P(x, y) : y[m] - x[m] = v$
                Predicate p = CreateRelativePredicate(m, other[m] - target[m])
                // create a condition of the form $f(x) = \mathcal{Q}y \in C : P(x, y)$,
                // where $\mathcal{Q}$ is any quantifier and $C$ is other.class
                Condition c = CreateQuantifiedCondition(p, other.class)
                conditions.add(c)
        // Add other-only predicates
        for (Member m in other.attributes) do
            // create a predicate of the form $P(y) : y[m] = v$
            Predicate p = CreateValuePredicate(m, other[m])
            // create a condition of the form $f(x) = \mathcal{Q}y \in C : P(y)$,
            // where $\mathcal{Q}$ is any quantifier and $C$ is other.class
            Condition c = CreateQuantifiedCondition(p, other.class)
            conditions.add(c)
    return conditions
```

Alg. 2. If any of the Candidates' scores becomes higher than the baseline (again, comparing using α confidence levels), those Candidates are moved to the `hypotheses` list and for each, a new Candidate is constructed by combining it with the current best hypothesis. This ensures that well-informed Candidates will be created regardless of the order that each Candidate is promoted to the `hypotheses` list.

We now provide more details on the helper classes. The FrequencyTable class supports two operations, `observe(x, y)` and `predict(x)`, with function signatures

- `observe(BitString x, Value y)` and
- `predict(BitString x) → ProbabilityDistribution<Value>`.

The FrequencyTable's `observe` function simply increments a counter in its (sparse) internal table corresponding to the given (x, y) position. To convert BitStrings and Values to table indices, BitStrings are interpreted as integers and every newly-seen output Value is assigned a unique integer id (e.g., by counting up from 0). As new pairs are observed, the table is dynamically resized when necessary. Given the table's counts, relative frequencies can be calculated and used to estimate probabilities (and the approximator's S score). The `predict` function calculates the conditional probability distribution $\hat{P}(y|x)$ of every output y for a given input value x based on this information.

The Condition class represents a function $f(o_1)$ constructed using quantified inner predicates (e.g., $f(o_1) = (\mathcal{Q}o_2 \in C_1 : o_2[pos] - o_1[pos] = (1, 0))$, where Q is a placeholder for any quantifier). It supports evaluation (which yields a bit string, as described in Section 3.2 of the paper) and addition (i.e., combining two formulas to produce a larger,

Algorithm 4: Evaluating Conditions

```
Func Condition.evaluate(State s, Object target) → BitString
    BitString result = ""    // initialize empty string
    // Construct a bit string containing the value of each relation group in the list;
    // this allows us to represent all logical connectives (between groups) at once
    for (RelationGroup g in my_relation_groups) do
        list<Predicate> predicates = g.predicates
        set<Object> objects = s.findObjectsOfClass(g.class)
        BitString value = EvaluateRelationGroup(predicates, target, objects)
        result.append(value)
    return result
Func EvaluateRelationGroup(list<Predicate> predicates, Object x, set<Object> others)
    $n$ = predicates.size()
    BitString result = BitString($2^n$)    // initialize string with $2^n$ zeros
    // Enable bits in result corresponding to every observed value of the predicates;
    // this allows us to represent all quantifiers at once
    for (Object y in others) do
        BitString value = EvaluatePredicateList(predicates, x, y)
        result[value] = 1    // reinterpret value as an integer and use it to index into the result
    return result
Func EvaluatePredicateList(list<Predicate> predicates, Object x, Object y) → BitString
    BitString result = ""    // initialize empty string
    // Construct a bit string containing the value of each predicate in the list;
    // this allows us to represent all logical connectives (within a group) at once
    for (Predicate p in predicates do
        boolean value = p.evaluate(x, y)    // predicates may ignore the second argument
        result.append(value)    // append a 0 or 1 to the bit string
    return result
```

more complex function). At a high level, evaluation operates as follows. First, assume that we have a Condition of the form $f(x) = (P_1(x) \odot P_2(x) \odot ...) \odot (\mathcal{Q}y \in C_1 : P_3(x,y) \odot P_4(x,y) \odot ...) \odot ...$ (where $\odot$ represents any possible logical connective). We will call each parenthesized group a *relation group*, which consists of a class (to bind variables from, possibly `null`, as with the first group in the example) and a list of predicates. To evaluate a relation group for some chosen y, we evaluate each inner predicate and create a binary string by concatenation. We then track which such strings (out of the 2^n possible) have been observed for all possible y in the group's class, again constructing a bit string that stores this information. Finally, the Condition's value is the concatenation of the bit string from each of its relation groups. Pseudocode for this process is in Alg. 4.

Addition of Condition objects is a more straightforward operation. Matching relation groups (i.e., referring to the same class type) in each are identified and their predicate lists are unioned to yield a larger list. For example,

$$(x[color] = (1)) \odot (\mathcal{Q}y \in C_1 : y[color] = (0)) \tag{19}$$

combined with

$$\mathcal{Q}y \in C_1 : y[position] - x[position] = (1, 0) \tag{20}$$

would yield the Condition

$$(x[color] = (1)) \odot (\mathcal{Q}y \in C_1 : (y[color] = (0)) \odot (y[position] - x[position] = (1, 0)))\,. \tag{21}$$

Prediction using QORA, shown in Alg. 5, is much simpler than learning. The Predictor simply uses its best hypothesis (which may be the baseline, if there is nothing better), as shown in Alg. 6.

A.1. Relation Group Visualization

In Figure 7, we show a visualization of the process of encoding a state observation into a relation group.

Algorithm 5: QORA Prediction Function

```
Func Qora.predict(State $s$, Action $a$) → State
    State $s'$ = State()    // initialize empty state
    for (Object obj in $s$) do
        Object obj' = Object(obj.class, obj.id)    // initialize empty object of same type with same id
        for (Member $m$ in obj.attributes) do
            Predictor p = predictors[obj.class, $m$, $a$]
            obj'[$m$] = obj[$m$] + p.predict($s$, obj)    // predict a distribution over this attribute's values
        $s'$.add(obj')
    return $s'$
```

Algorithm 6: Predictor Prediction Function

```
Func Predictor.predict(State $s$, Object target) → ProbabilityDistribution<Value>
    if (hypotheses is empty) then
        return baseline.predict(0)
    else
        Candidate h = hypotheses[0]    // best hypothesis
        BitString c = h.condition.evaluate($s$, target)
        return h.table.predict(c)
```

B. Benchmark Environments

We describe four testing domains, ranging from simple to complex. Although QORA could be applied to more-complex environments, these were designed to focus on particular aspects of an algorithm's learning and demonstrate problems with prior work while being easy for humans to understand. Each domain has several parameters (e.g., width, height) that control the initial states it will generate. Note that these settings have no effect on the behavior of the environment, i.e., T; therefore, by changing the parameters from training to test time, we can evaluate a learner's knowledge transfer capabilities. Effective generalization is facilitated by learning *rules*, which are distinct components of the model that typically require a relatively small amount of information to make predictions (e.g., "when the move-right action is taken, the player moves to the right unless blocked by a wall").

Walls Our first domain, `walls`, is a baseline for the ability to learn relational rules. The two classes of object, `player` and `wall`, both only have `position` attributes. There are five actions: movement in each direction (`left`, `right`, `up`, `down`) and `stay`. If a movement action is chosen, the player will move one unit in the intended direction unless there is a wall blocking the position it would move to; if the `stay` action is taken, nothing happens. Walls are never modified by any action. Example states are shown in Figs. 8a and 8b.

Though this domain sounds simple from a human perspective, it is important to highlight the difficulty that arises when a learner has no prior knowledge of the world. If we must learn T from scratch, making only weak assumptions about its form, the sheer amount of information contained in even an 8×8 world of the `walls` environment is immense. It is *possible*, for example, that the player's movement is determined not by the presence of adjacent walls but by some convoluted function based on its position and the presence of some arbitrary set of walls scattered throughout the world. Though it may seem obvious to a human, the learner has no way to rule this out *a priori*. Thus, as there are almost 300 basic facts (e.g., "the player is at position (x, y)") that can be observed in an 8×8 level (36 player positions within the world's border, 64 wall positions, and 13^2 possible player-wall relative positions), and the number of possible formulas is super-exponential in this number, searching through the entire space of conditions is intractable. Previous works have made strong assumptions Hershkowitz et al. (2015) or required the correct formulas to be given for each domain (Diuk et al., 2008; Marom & Rosman, 2018), significantly reducing the difficulty of their problems while also reducing the generality of their solutions.

Lights The `lights` domain is a relational test *not* involving a grid-structured world. Instead, there is a single `switch` object and several `lights`. All objects have an `id` and each light is either on or off. The switch's id can be changed using the new `increment` and `decrement` actions. When the new `toggle` action is taken, any light with the same id as the switch will toggle its state.

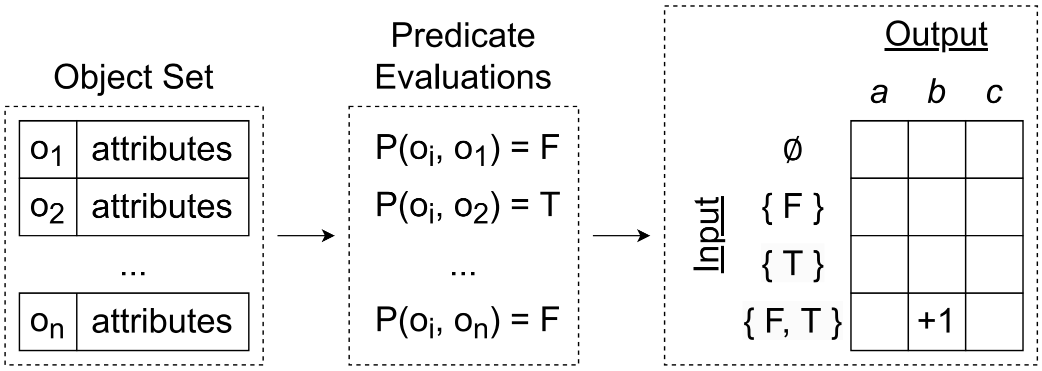
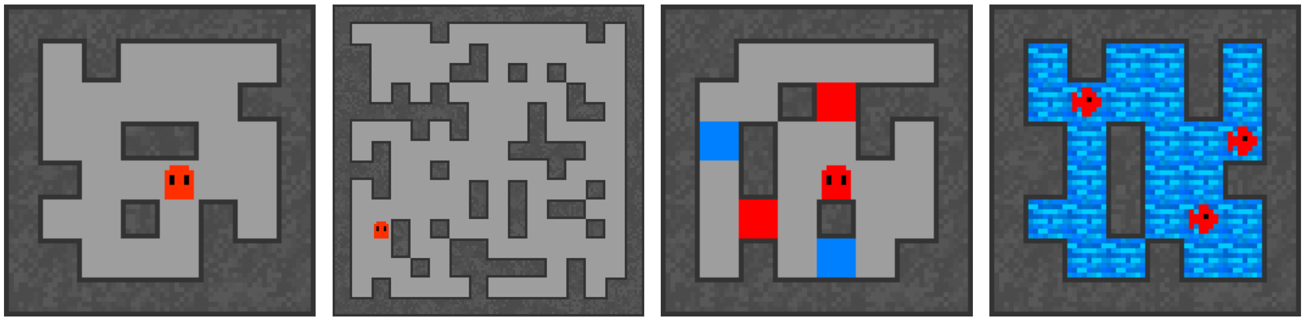


Figure 7: Illustration of a relation group containing predicate P being evaluated with respect to some object o_i. The value of P is computed for each pair (o_i, o_j) and the values (T, F) that appeared are tracked. In this case, both values (T and F) are present, so we increment the table under the corresponding output for this observation. In this example, the output is b; this is determined by a separate process.



(a) an 8×8 level from the `walls` domain; the player character is denoted by the red icon

(b) a 16×16 level in the `walls` domain

(c) a level from the `doors` domain; doors are denoted by the red and blue squares

(d) a level from the `fish` domain

Figure 8: Example levels from three domains: (a-b) walls, (c) doors, and (d) fish. In (c), the player is currently able to pass through red doors; blue doors will block their movement until they change color.

Doors This domain adds significant complexity to the types of rules that the agent must learn. In this environment, an extension of the `walls` domain, there is an additional object class called `door` and an additional member attribute called `color` that takes values in $\{0, 1\}$. The player and doors each have a color. Doors act like walls, except that the player can step onto doors that share its color. There is also a new action called `change-color` that allows the player to swap its color when not standing on a door. As an example of the formalism described in Section 2, the types in the `doors` environment can be expressed as:

$$\begin{aligned} M &= \{(\text{position, 2}), (\text{color, 1})\} \\ C &= \{(\text{player}, \{\text{position, color}\}), (\text{wall}, \{\text{position}\}), \\ &\quad (\text{door}, \{\text{position, color}\})\} \\ A &= \{\text{STAY, MOVE\_LEFT, MOVE\_RIGHT, MOVE\_UP,} \\ &\quad \text{MOVE\_DOWN, CHANGE\_COLOR}\} \end{aligned}$$

Fish The `fish` domain is designed to test an agent's ability to learn stochastic transitions. In this environment, there are walls and some number of "fish". The agent can take two actions: `stay`, which does nothing, and `move`, which causes each fish to move around randomly. Specifically, each fish will choose a direction uniformly at random from {left, right, up, down} and move one unit in that direction unless blocked by a wall. To fully solve this domain, an agent must learn the conditional probability of each movement direction based on the existence of surrounding walls; i.e., it must be able to generate a *distribution* over future states.

B.1. Domain Sets

To test QORA's ability to tackle more-complex tasks, we use parameterized sets of domains. We consider these to be sets of domains, rather than just single domains, due to the parameterization affecting the transition function T. Each set generalizes the `walls` domain in some way.

Moves The `moves` domains add an arbitrary number of copies of each movement action, each applying to the singular player object. Thus, although the number of actions varies, the complexity of the environments does not change in a significant way.

Players The `players` domains add an arbitrary number of players. Each player is controlled by its own set of four independent actions, meaning that the number of actions is four times the number of players.

Paths The `paths` domains generalize the `walls` domain to an arbitrary number of dimensions n. To save computational resources, instead of being blocked by walls, the player in these domains can only move onto grid spaces occupied by `path` objects. Thus, interesting levels can be generated without requiring a number of objects $\propto 10^n$.

Complex The `complex` domains are five domains that incrementally add novel, challenging interactions for the learner to model. `complex`(0) is the walls domain. `complex`(1) adds `gate` objects that block the player's movement, similarly to walls. `complex`(2) adds a `guard`, along with four new move actions to control it, that can move through gates but not through walls. `complex`(3) adds `switch` objects that toggle their state when the player moves over them. `complex`(4) adds directional `jump` actions that allow the player to vault over `gate` objects (but only if the other side of the gate is not blocked). To help the random-policy agent gather data more efficiently, the domain is configured to typically start the player near a gate; if not for this fact, the frequency with which the agent experiences "interesting" jumps (i.e., not just trying to jump into a wall or empty space, both of which are no-ops) would be vanishingly small, therefore not allowing the learner's data-efficiency to be demonstrated.

C. Neural Network Baseline Architectures

Our implementation of the NPE is shown in Figure 9. It is based on the description in (Chang et al., 2016) with the addition of the pre-encoding network *f*. The particular architectures of *f*, *g*, and *h* are our own, based on several iterations of network design improvement. The total number of parameters is in this structure is 6,944. We tried more complex architectures, including other ways of encoding output (e.g., using attribute deltas like QORA), without noticeable improvement.

The MHDPA baseline consisted of a single 5-head dot-product attention layer (as implemented by PyTorch) followed by a per-object encoder similar to the *h* module in the NPE that appends the action to each object. The total number of parameters in this structure is 2,784. We tested more complex architectures (e.g., a pre-encoder), but they only led to slower learning without improving accuracy.

For both NPE and MHDPA, we used 10 batches of 100 observations per epoch. The learning rate was 0.01 for NPE and 0.005 for MHDPA. We used stochastic gradient descent with momentum of 0.9 and L1 loss (since outputs were linear).

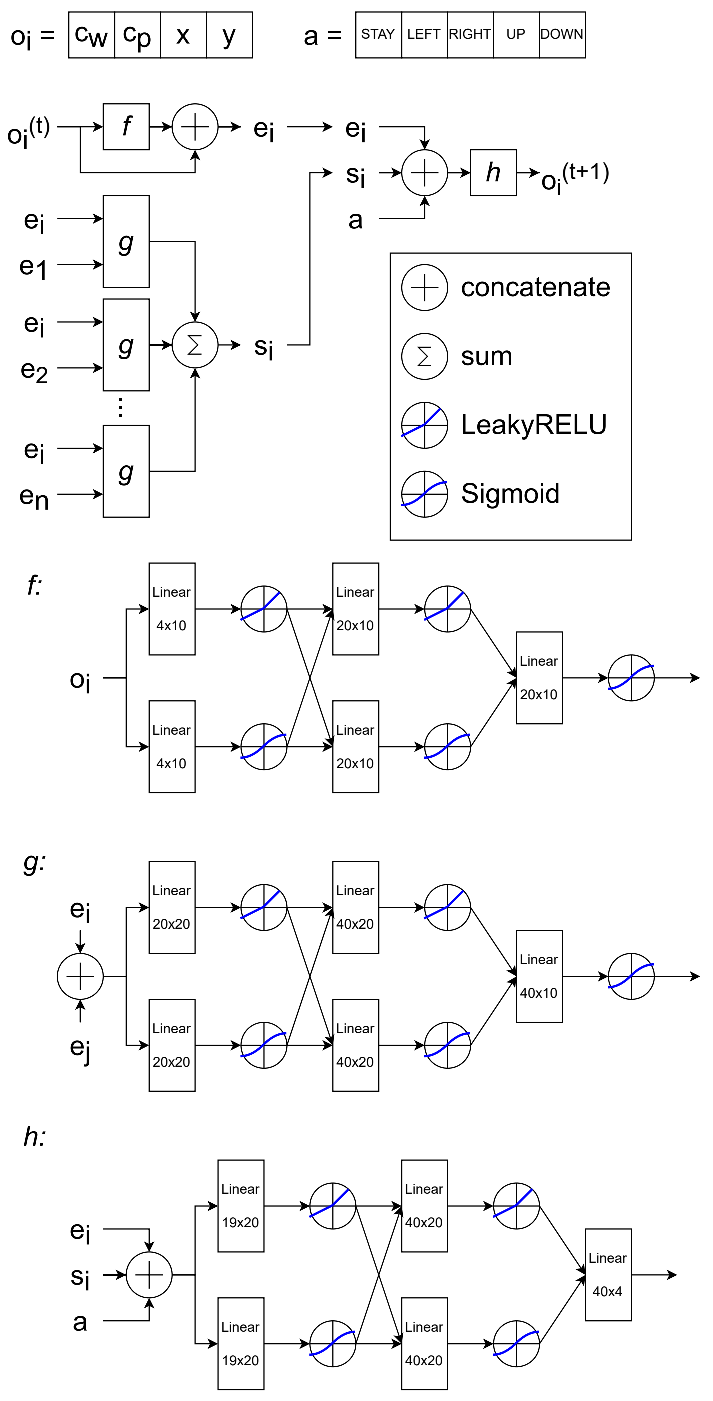


Figure 9: Architecture of our NPE baseline. Let o_i be an object, which is encoded with four attributes: its one-hot class (wall or player, c_w or c_p respectively) and its position (x and y coordinates). Let a be the current action, which is one-hot encoded. Objects go through a pre-encoding stage where features can be extracted (f) followed by a pairwise computation (g) and a final output calculation stage (h). More details can be found in Chang et al. (2016).