Learning Integrated Relational and Continuous Action Models for Continuous Domains

by

Joseph Zhen Ying Xu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2013

To my parents.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Learning Integrated Relational and Continuous Action Models for Continuous Domains

by

Joseph Zhen Ying Xu

Chair: John E. Laird

Long-living autonomous agents must be able to learn to perform competently in novel environments. One important aspect of competence is the ability to plan, which entails the ability to learn models of the agent's own actions and their effects on the environment. This thesis describes an approach to learn action models of environments with continuous-valued spatial states and realistic physics consisting of multiple interacting rigid objects. In such environments, we hypothesize that objects exhibit multiple qualitatively distinct behaviors based on their relationships to each other and how they interact. We call these qualitatively distinct behaviors *modes.* Our approach models individual modes with linear functions. We extend the standard propositional function representation with learned knowledge about the *roles* of objects in determining the outcomes of functions. Roles are learned as first-order relations using the FOIL algorithm. This allows the functions modeling individual modes to be "instantiated" with different sets of objects, similar to relational rules such as STRIPS operators. We also use FOIL to learn preconditions for each mode consisting of clauses that test spatial relationships between objects. These relational preconditions naturally capture the interaction dynamics of spatial domains and allow faster learning and generalization of the model. The combination of continuous linear functions, relational roles, and relational mode preconditions effectively capture both continuous and relational regularities prominent in spatial domains. This results in faster and more general action modeling in these domains. We evaluate the algorithm on two domains, one involving pushing stacks of boxes against frictional resistance, and one in which a ball interacts with obstacles in a physics simulator. We show that our algorithm learns more accurate models than locally weighted regression in the physics simulator domain. We also show that relational mode preconditions learned with FOIL are more accurate than continuous classifiers learned with support vector machines and k-nearest-neighbor.

CHAPTER I

Introduction

1.1 Action models

Action models are models of how an agent's actions change the environment. Let $\mathbf{x_t}$ represent the state of the environment at time t, and $\mathbf{u_t}$ represent the agent's action at time t, as shown in figure 1.1. An action model is a function that predicts the next state of the environment given the current state and action:

$$F(\mathbf{x_t}, \mathbf{u_t}) \rightarrow \mathbf{x_{t+1}}$$

Accurate action models have many uses for intelligent agents. They allow the agent to simulate the outcomes of actions without the cost of executing them in the real world. Simulating actions is usually faster and safer than executing them. For example, taking exploratory actions such as driving off a cliff does not incur any penalties in simulation. Simulating extended plans also allows the agent to backtrack quickly when it runs into dead ends, which is often not possible in the real world.

AI planning algorithms rely on action models represented as STRIPS operators (*Russell and Norvig*, 2003) or PDDL operators (*Ghallab et al.*, 1998). In robotics, motion planning algorithms such as Rapidly-exploring Random Trees (RRT) (*LaValle and Kuffner*, 2001) require models of continuous-valued actions and state changes.

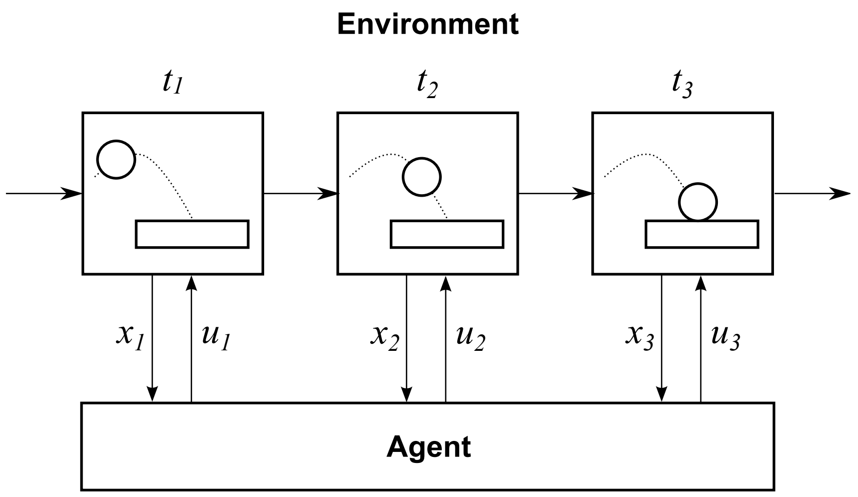


Figure 1.1: Environment setup with discrete time steps.

Action models have also been used to speed up reinforcement learning by giving the agent "random access" to any part of the state-action space for doing temporal difference backups (*Sutton*, 1991; *Moore and Atkeson*, 1993).

In some applications, the agent designer understands the environment well enough to hand-code accurate action models into the agent's knowledge. However, there are other applications in which this is not possible. For example:

- The agent may have to deal with novel environments that cannot be anticipated by the agent designer. This is the case for long-living, situated agents that will encounter a variety of tasks and environments.
- There are too many variations in the action model to enumerate by hand. For example, the number of interactions in an environment with a large number of objects will be combinatorially large.
- The agent's sensors and effectors may change in unexpected ways. For example, a robot's motors will behave differently as energy is consumed.
- A model of the agent's effectors is too complicated for a programmer to express in code. This is the case for robots with many degrees of freedom.

In these applications, one possibility is that the agent learn action models by observing the effects of its actions in the world. Learning an action model can be formulated as learning the function $F(\mathbf{x_t}, \mathbf{u_t}) \rightarrow \mathbf{x_{t+1}}$ from a sequence of observations in the form of state-action pairs

$$\mathbf{x_1}, \mathbf{u_1}, \mathbf{x_2}, \mathbf{u_2}, \ldots$$

where each adjacent pair of state observations and action forms a training example

$$(\mathbf{x_t}, \mathbf{u_t}, \mathbf{x_{t+1}})$$

The appropriate model learning algorithm to use depends to a large degree on the characteristics of the environment being modeled and how the state $\mathbf{x}_t$ is represented. $\mathbf{x}_t$ can be a set of relations such as $\mathit{ontop}(A, B)$ and $\mathit{clear}(\mathit{table})$ in the classic blocks world, a vector of real numbers encoding the positions and rotations of joints in a continuous robotics domains, or a combination of both. Likewise, $\mathbf{u}_t$ can be a discrete action such as `move-block(A,C)`, or a vector of real numbers representing motor voltages to send to each joint in a robot arm.

We are interested in the problem of learning action models in spatial, object-based environments. These are two or three dimensional environments in which multiple rigid-body objects move and interact based on physical laws. In the most general case, agent actions in the environment are also continuous, such as voltages to output to a robot's motors. Many real world environments can be characterized in this way.

Relational action modeling approaches such as those that learn STRIPS-style operators(*Wang*, 1995; *Carbonell and Gil*, 1996; *Kaelbling et al.*, 2007) are usually insufficient for such environments because they don't model continuous properties and actions. On the other hand, most continuous action modeling techniques are propositional and not sufficiently expressive when the environment has multiple interacting objects. For example, when a ball bounces down a set of stairs, the same bouncing interaction recurs between the ball and a different step each time. Intuitively, a single model should be able to describe all the bounces, regardless of the step involved. This requires that the model be first-order, so that instead of modeling a specific ball and step, it models abstract ball and step "variables", which can then be "instantiated" with any ball and step. We present an algorithm for learning models that are both relational and continuous, and can capture these types of regularities.

This thesis focuses on learning models of action outcomes and environment dynamics, but does not deal with action selection. We assume that actions are generated by another source, such as a separate decision module in the agent. Therefore, our models do not have an explict concept of actions. Instead, actions are encoded as regular properties of the state (see section 1.2.3), and our action models are formally the same as environment models. This is consistent with the notion of action modeling in the robotics literature (*Nguyen-Tuong and Peters*, 2011). The interactions between action selection and model learning should be considered in future work, but we ignore it for now to focus on modeling environment dynamics.

For the rest of this chapter, we will describe in detail the characteristics of the environments we consider in this thesis. We then enumerate a set of requirements for learning action models in these environments, and finally give an overview of our approach.

1.2 Environment characterization

Many artificial and real-world environments can be characterized as a set of interacting rigid objects with fixed shapes in 3D space. For example, in a realistic version of blocks world, the objects are the gripper, the blocks, and the table. The interactions are the gripper picking up and dropping blocks, one block or the table supporting a block on top of it, and collisions between all the objects. Other traditional AI tasks

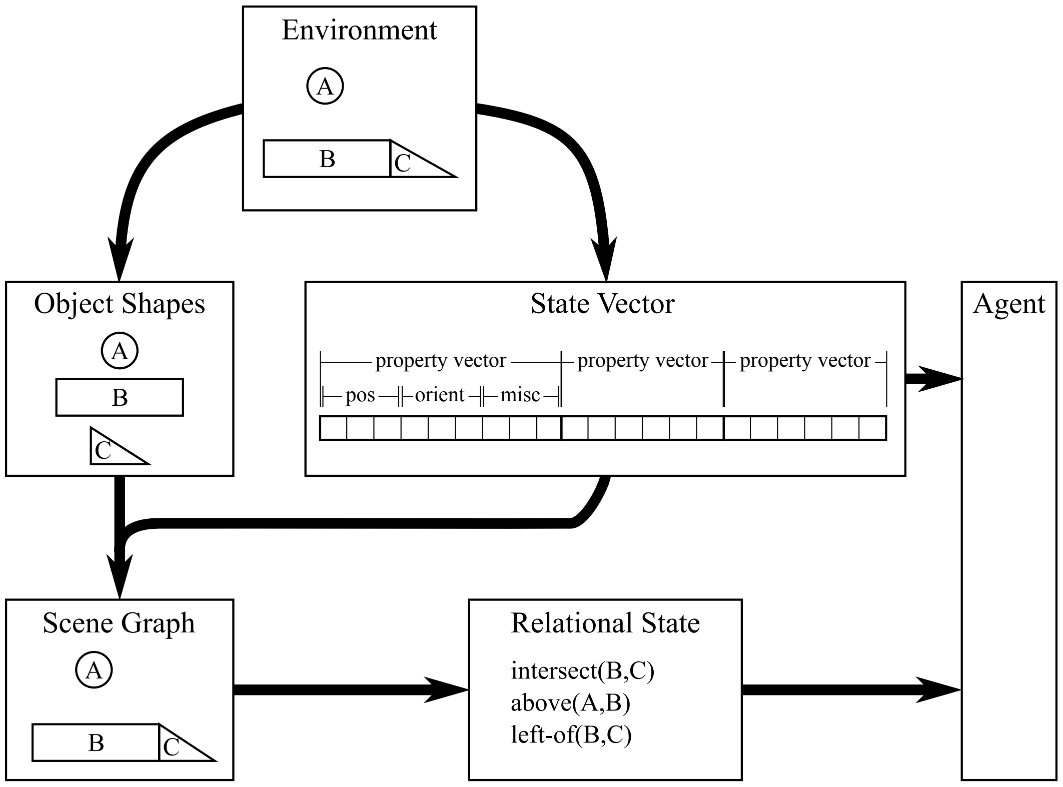


Figure 1.2: State representation.

such as Towers of Hanoi, 8-puzzle, N-queens, etc. fit this characterization. So do more complex domains such as automatic car driving in an urban environment.

There are many ways to represent these types of environments. Traditional AI research used relational representations, such as the classic blocks world, where the state is encoded as a set of *ontop*(X, Y) relationships. A strength of this representation is that task-relevant information such as the *ontop* relationship can be encoded compactly. Expressing the *ontop* relationship in a different representation, such as one that encodes the (x, y, z) positions of blocks, is complex and not general. On the other hand, a major weakness of this representation is that it cannot encode the continuous information of the environment, such as the actual positions of the blocks.

On the opposite end of the spectrum is the propositional, continuous representation. The state in this representation is a vector of real numbers, each representing a continuous dimension such as the x position of one of the blocks. The strength of this representation is that it naturally encodes continuous properties of the environoment. A weakness is that some high-level task-relevant information, such as the spatial relationships between blocks, is only implicitly captured.

Our system uses a combination of both representations in order to get the best of both worlds. Figure 1.2 gives an overview of our state representation. We assume that the environment is composed of a set of objects in 3D space. Each object has a position, orientation, and shape. Objects can also have other arbitrary properties such as velocity, density, friction coefficient, etc. that can be encoded as real numbers. Each object has a type, and objects of the same type have the same set of properties. Object shapes are encoded as the vertices of a convex polyhedron or parameters of geometric primitives such as the radius of a sphere.

1.2.1 Continuous State

The position, orientation, and miscellaneous properties of an object are concatenated into a vector of real numbers, which we call a *property vector*. We require that objects of the same type have property vectors of the same length, and that the order of the properties in the vector is the same. The property vectors for all objects in the environment are concatenated into the *state vector*, such that all properties of a single object are contiguously grouped. The number and order of the property vectors in the state vector can change as objects appear or disappear from the environment, as well as when the agent enters a different environment. An updated state vector is provided by the environment to the agent at each time step. Object shape information is not included in the property vector and is provided to the agent through a separate channel. We assume that object shapes do not change over time, so this information is only given to the agent once.

The coordinate system used to encode object positions and orientations is important for model learning since it contains implicit biases and invariances that affect the difficulty of learning certain types of models, as well as the generality of the learned models. For example, using polar coordinates to encode the positions of the planets relative to a star makes it easier to learn a model of planetary motion, since the distance to each planet never changes (assuming circular orbits), and the angles change by a constant amount per unit time. On the other hand, using polar coordinates to encode the positions of chess pieces relative to the king makes it difficult to model their movements. Since most pieces move on axes aligned with the board and not radially aligned with the king, the movements would result in complex changes in angle and distance. In this case, rectangular coordinates relative to a corner of the board results in simpler and more general models.

Because we want our model learning algorithm to work in a wide variety of environments, we do not require object positions and orientations to be encoded in a particular coordinate system. However, the agent must be able to transform the position and orientation coordinates into absolute rectangular coordinates and Euler angles, respectively. This is so that the agent can construct a scene graph from the state vector and shape information, from which it extracts spatial relationships between objects. Note that the model function learning algorithm still operates over the original position and orientation representation, and will still exploit the invariances and biases of that representation.

1.2.2 Relational State

As mentioned previously, a weakness of propositional continuous representations is that task-relevant information is only implicitly encoded. In spatial environments, the relationships between objects often determine how they interact, and is therefore important to the action model. To make this information explicit, we assume that our system is able to compute the truth values of a set of spatial relations for the objects in the environment. Some example relations include $intersect(X, Y)$, $ontop(X, Y)$, and *east*(X, Y), where X and Y are objects in the environment. The set of relations is fixed across all environments. We call the exhaustive list of these relations at time t the *relational state* at time t, abbreviated as $\mathbf{r}_t$. We discuss how our system calculates these relations in section 3.3.

The set of relationships was chosen to maximize applicability to a wide variety of environments. For example, most types of physical interactions between rigid objects require the objects to be touching, such as collisions, friction, and support. Therefore, the *intersect* relation is in the set. Since gravity is ubiquitous in physical environments, relations such as *above* and *ontop* are also included. Of course, not all important relations can be anticipated a priori. Some must be learned from experience in a specific environment. We do not attempt to solve this problem in this thesis, but discuss a possible approach in section 3.4.4.

The relational state in our system only encodes information that is implicitly present in and can be derived from the continuous state. For example, the truth of *ontop*(A, B) can be calculated from the positions and geometries of A and B, which are encoded in the continuous state. This is different from some hybrid representations in which the relational and continuous data encode orthogonal information. For example, in probabilistic relational models (*Getoor and Taskar*, 2007), a relation may encode whether a student is enrolled in a class, while a continuous attribute may encode the student's GPA. Our system does not handle abstract relations such as class enrollment.

1.2.3 Action modeling in this representation

As mentioned previously, the environment state $\mathbf{x}_t$ is encoded as a vector of real numbers. We assume that the agent's actions are represented as vectors of real numbers, such as the voltages to apply to each joint in a robot arm. The action vector is encoded as properties of a virtual object with no physical body, so actions are simply considered additional properties in the state vector. The models our system learns are conditioned on these action properties just like any other state property; actions are not treated specially. Therefore, we rewrite the model function $F(\mathbf{x}_t, \mathbf{u}_t)$ as just $F(\mathbf{x}_t)$ with the assumption that $\mathbf{u}_t \subset (\mathbf{x}_t)$.

To simplify the problem further, we assume that each dimension of the state vector $\mathbf{x}_{t+1}$ can be the modeled independently. Therefore, $F(\mathbf{x}_t) \to \mathbf{x}_{t+1}$ can be decomposed into a set of simpler functions, each of which predicts a *single* dimension of $\mathbf{x}_{t+1}$.

$$\{F(\mathbf{x}_t) \to y_1, F(\mathbf{x}_t) \to y_2, \ldots, F(\mathbf{x}_t) \to y_k\}$$

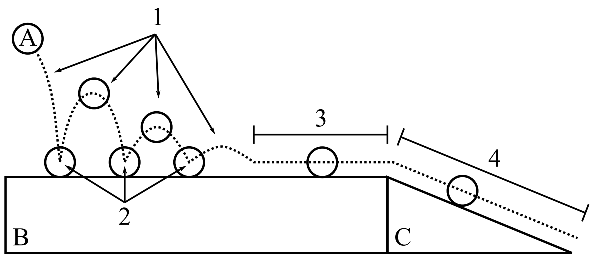
where k is the size of the state vector, and $y_1 = \mathbf{x}^1_{t+1}, y_2 = \mathbf{x}^2_{t+1}, \ldots, y_k = \mathbf{x}^k_{t+1}$ are the individual elements of the state vector. In this thesis, we focus on learning these individual models. We call y_i the *target property* of the model, and the object that the property belongs to the *target object*. To learn a complete model, the agent must independently learn a model for each state dimension. While this may be inefficient when multiple state dimensions are related, it significantly simplifies the model learning algorithm, and does not hurt the generality of the results.

1.2.4 Influences and behavioral modes

Environments with discrete interacting objects have two broad classes of influence on the behaviors of those objects. We call the first class *internal influences*. For example, if a wheeled robot is treated as an atomic object, then its ability to convert energy stored in its battery into driving movements is a type of internal influence. Behaviors caused by this class of influence are usually conditioned only on the internal states of objects, such as the voltages applied to the drive motors of the robot, or the momentum of the robot's motion. The second class of influences are *external influences* and come from interactions between objects. An example of an external influence is a robot driving into a wall: the wall blocks the robot's forward movement by exerting an equal force on it in the opposite direction. Another example would be the robot driving over different types of terrain, which affects its movement and turning behaviors. Behaviors caused by external influences are conditioned on the relationships between objects, such as whether the robot touches the wall.

The behavior of an object is determined by a combination of internal and external influences. For example, a ball flying through the air is only subject to the influence of gravity and momentum, while a ball rolling down a ramp is subject to gravity, momentum, rolling friction, and the force of the ramp pushing up against it. Each unique combination of influences gives rise to a possibly unique behavior. The ball flying through the air follows a different trajectory than when it rolls down the ramp. When the set of influences affecting an object is constant, the object tends to behave smoothly. On the other hand, changes in influences tend to cause abrupt, unsmooth behaviors. The ball flying through the air follows a smooth arc, but when it hits the ramp, its trajectory undergoes a sudden change in direction. Both internal and external influences can change abruptly. Internal influences may change due to some change in operation, such as a brake being engaged or a gearbox being switched to reverse. External influences usually change when an object enters different relationships with other objects, such as when the ball transitions from flying to rolling upon contact with the ramp. We will call the distinct behaviors that arise from different combinations of influences *modes*. Modes are a characterization of overall behavior, not the individual influences or causes of a behavior.

Complex behaviors resulting from object interactions can often be decomposed into a sequence of simpler modes. Consider the scenario in figure 1.3. Suppose that the system is learning a model of the vertical component of the velocity of ball A (v_y). The trajectory of the ball is complex and difficult to model as a whole, but it can



#	Mode	Influences	Relations	Function
1.	Flying	Gravity	~intersect(A, *)	$v_y' = v_y - k_1$
2.	Bounce (flat)	Gravity, surface pushing up	intersect(A, B) & $v_y > 10^{-8}$	$v_y' = -k_2 v_y$
3.	Roll (flat)	Gravity, surface pushing up, friction	intersect(A, B) & $v_y \leq 10^{-8}$	$v_y' = 0$
4.	Roll (ramp)	Gravity, surface pushing up, friction	intersect(A, C)	$v_y' = k_3 v_x + k_4 v_y + k_5$

Figure 1.3: Modes, influences, and behavior of a ball bouncing on a box and ramp.

be broken down into a sequence of four modes conditioned on interactions between the ball (A), the box (B), and the ramp (C). As shown in the figure, each mode has a different set of influences, resulting from different relationships between the ball and the other objects. The influences column lists which physical forces influence the behavior of the ball in each mode. Note that each mode describes a combination of forces and does not try to analyze how each individual force contributes to the overall behavior of the ball. Such an analysis would require extensive background knowledge about the laws of physics that our system does not possess.

Each entry in the relations column can be thought of as the preconditions that must hold for the corresponding mode to be exhibited. Therefore, the relations that hold at a particular time can be used to anticipate the mode that will be active at that time. The precondition of the flying mode is that the ball (A) is not intersecting any other objects. The precondition for the bouncing mode is that the ball intersects the flat platform (B) and its y velocity is above a small threshold (10^{-8}). This threshold is the value at which bouncing damps out to rolling, and depends on factors such as floating-point round-offs in a physics simulator, or the sensitivity of sensors in detecting the difference in the vertical position of the ball. The exact value of this threshold varies from system to system. When the ball's y velocity is below the threshold, then it is in the flat rolling mode. Finally, the precondition for the ramp rolling mode is that A intersects the ramp (C).

The last column in the table shows the y velocity at time $t+1$ (v'_y) as a function of the y velocity at time t (v_y) for each mode. These functions are much simpler than a single function that describes the entire trajectory of the ball. Furthermore, each function is only conditioned on the local objects involved (e.g. only the ball and the object it contacts), and can be applied to any interaction of the same type, regardless of where the interaction occurs or how other objects are arranged. We call these functions *mode functions*.

Decomposing system behavior into qualitatively distinct parts has been studied in the qualitative reasoning literature. Modes are closely related to the idea of *processes* in qualitative process theory (*Forbus*, 1984). A qualitative process defines a set of relevant objects, preconditions on the properties of those objects, and how object properties affect each other. Effects on properties are specified as a relationship between two properties and a direction of change, such as "the rate of heat flow from a source to a destination decreases as the temperature of the destination increases." The primary difference between modes and processes is that modes model the behavior of a property as an aggregation of all influences, while processes model individual influences. For example, if a box is pushed on a rough surface, its velocity is influenced by both the pushing force and the frictional force. This situation would be described by a single mode (that of being pushed on a rough surface) but two processes (one for being pushed, and one for frictional force).

QSIM (*Kuipers*, 1994) decomposes whole system behavior into a set of *operating regions* which are analogous to modes. Within each region, the behavior of the system is described by qualitative differential equations (QDEs), which are analogous to mode functions. Transitions between regions are defined by transition functions that test properties of the system. This is different from our characterization of modes which are conditioned on the true relationships at every time step.

1.2.5 Role identification and assignment

A mode function can be conditioned on the properties of several objects, and each object affects the behavior of the mode in a different way. In other words, each object plays a distinct role in determining the behavior of a mode. Consider an elastic collision between two balls, with masses m_A and m_B. The function for the exit velocity v'_A of the ball with mass m_A is

$$v'_A = \frac{m_A - m_B}{m_A + m_B} v_A + \frac{2m_B}{m_A + m_B} v_B$$

where v_A and v_B are the pre-collision velocities of the balls. Let $k_1 = \frac{m_A - m_B}{m_A + m_B}$ and $k_2 = \frac{2m_B}{m_A + m_B}$. Then the function is simply

$$v'_A = k_1 v_A + k_2 v_B$$

There are two roles in this mode: role A for the ball with mass m_A and role B for the ball with mass m_B. Each role is uniquely identified by the terms in the equation it participates in. The velocity of the ball fulfilling role A is multiplied by k_1 while the velocity of the ball fulfilling role B is multiplied by k_2.

This mode function can be used to model any elastic collision between any two balls with masses m_A and m_B. In this sense, roles can be thought of as abstract placeholders or variables, and the objects that fulfill each role can be thought of as an instantiation of that variable. We observe that in spatial domains, simple behaviors that are repeated across analogically similar situations are ubiquitous. Conditioning mode functions on abstract roles instead of actual objects is therefore crucial to learning general models. However, the correct objects must be assigned to the roles in order for the mode function to be correct. For example, using the velocity of a ball not involved in the collision, or switching the roles of the two balls in the above function would result in incorrect predictions. Even if a function perfectly describes an interaction, it will make an incorrect prediction if its roles are fulfilled by the wrong objects. Therefore, the system must identify the appropriate object to fulfill each role. We will call this problem the *role assignment problem*.

The dual to the role assignment problem is the role identification problem. When learning a model of a novel environment, the learning algorithm does not know a priori which roles exist. Not knowing the roles also means not knowing how the attributes in the training data should be aligned when applying a regression algorithm such as linear least squares. If we consider model learning as an optimization problem that minimizes the prediction error of the model, the role identification problem can be thought of as extending the optimization to search over all possible permutations of the attributes of each training example. Learning general models requires that the system solve both the role identification and assignment problems. We will address these problems directly in our approach.

1.2.6 Summary of Assumptions

Here we summarize the assumptions we make about the environment and model learning problem.

- The environment is composed of multiple interacting rigid objects with fixed shapes in 3D space.

- Each object has a type. Objects of the same type have the same properties and behave identically.

- Object shapes are described as convex polyhedrons or geometric primitives such as spheres.
- Object positions can be encoded in absolute rectangular coordinates. Object orientations can be encoded in Euler angles. The position and orientation of an object along with other continuous properties are composed into a property vector.
- The state of the environment excluding object shapes can be encoded as a concatenation of all property vectors, called the state vector. The value of each element of the state vector at time $t + 1$ is deterministic given the state vector and object shapes at time t, but can be degraded by Gaussian noise.
- Agent output to the environment can be encoded as a vector of real numbers. The model learner will consider the output vector as a component of the state vector.
- A model for each dimension of the state vector can be learned independently. The prediction of the next environment state $\mathbf{x}_{t+1}$ can be made by composing independent models of each dimension $F(\mathbf{x}_t) \to y$, where $y \in \mathbf{x}_{t+1}$.
- The agent can compute the truth values of a set of spatial relations given the state vector and object shapes. The set of spatial relations is fixed across environments.
- The behaviors of individual modes can be modeled accurately by linear functions.

1.3 Requirements for model learning

We enumerate some requirements for a model learning algorithm to perform well in environments with the characteristics previously described.

R1 The accuracy of the model should not be affected by irrelevant objects and properties. If the model predicts the trajectory of a ball, it shouldn't depend on objects the ball does not contact, unless they have influences over the ball's behavior via "action at a distance" such as gravity or magnetic fields. Similarly, a model of free fall should not depend on the friction coefficient of the object, since it has no effect on the outcome. A model learned from an object with one friction coefficient should be applicable to an object with a different friction coefficient.

R2 The accuracy of the model should not be affected by object location coordinates that are based on arbitrary origins. In spatial domains, only the relationships between objects matter. A model learned for a falling ball should not depend on whether the ball's y coordinate is 1.0 or 5.0, only the difference between the y coordinates of the ball and the ground. This invariance can be achieved by centering object positions on a behavior relevant point, such as the center of the ball, but this strategy isn't immune to other types of variations, such as balls with different radii.

R3 A model of an interaction between several objects should correctly predict the same interaction in any other set of objects, as long as they have the correct types and relationships, and regardless of other irrelevant objects in the environment. This requires that models be agnostic to object names and should not rely on them to identify object roles.

R4 The model should be able to represent abrupt and discontinuous changes in behavior that can occur at the boundaries between different sets of external influences. For example, a ball transitioning from falling to bouncing undergoes an abrupt change in its direction of travel.

R5 The model should be learned online as opposed to in batch. This means that the agent does not have random access to the environment. For example, a robot has a definite position in the world, and can only move within a relatively small neighborhood with each action. Furthermore, the agent may need to use a partially learned model to plan a route to other areas of the state space so that it can obtain more training examples. Therefore, the algorithm cannot wait until it has a representative distribution of training examples before constructing a useful model. It should try to utilize all available examples as soon as possible to incrementally improve the model's accuracy. It should also be able to incorporate future training examples without lowering prediction accuracy for the parts of the space it previously learned.

1.4 Approach Overview

Figure 1.4 shows an overview of our model learning algorithm. Our approach is based on the observation that learning separate models of individual behavioral modes is simpler and more general than learning a single model of the total behavior of the environment. Toward this end, our model learning algorithm must solve two major problems, corresponding to the two major components in our system. The first problem is to discover the modes in the system and the linear functions that describe their behavior. This component is labeled as "clustering" in the figure. The second problem is to discover a set of rules about how to compose the individual modes into a complete model. This component is labeled as "classification" in the figure.

The clustering component of our system tries to identify behavioral modes in the training data by clustering examples based on similarity. This component works at the level of the continuous state vector representation. We define two training examples to be similar if they fit the same linear function. The end product of this component is a grouping of training examples into clusters, with all examples in a single cluster fitting the same linear function. The rationale is that these clusters will

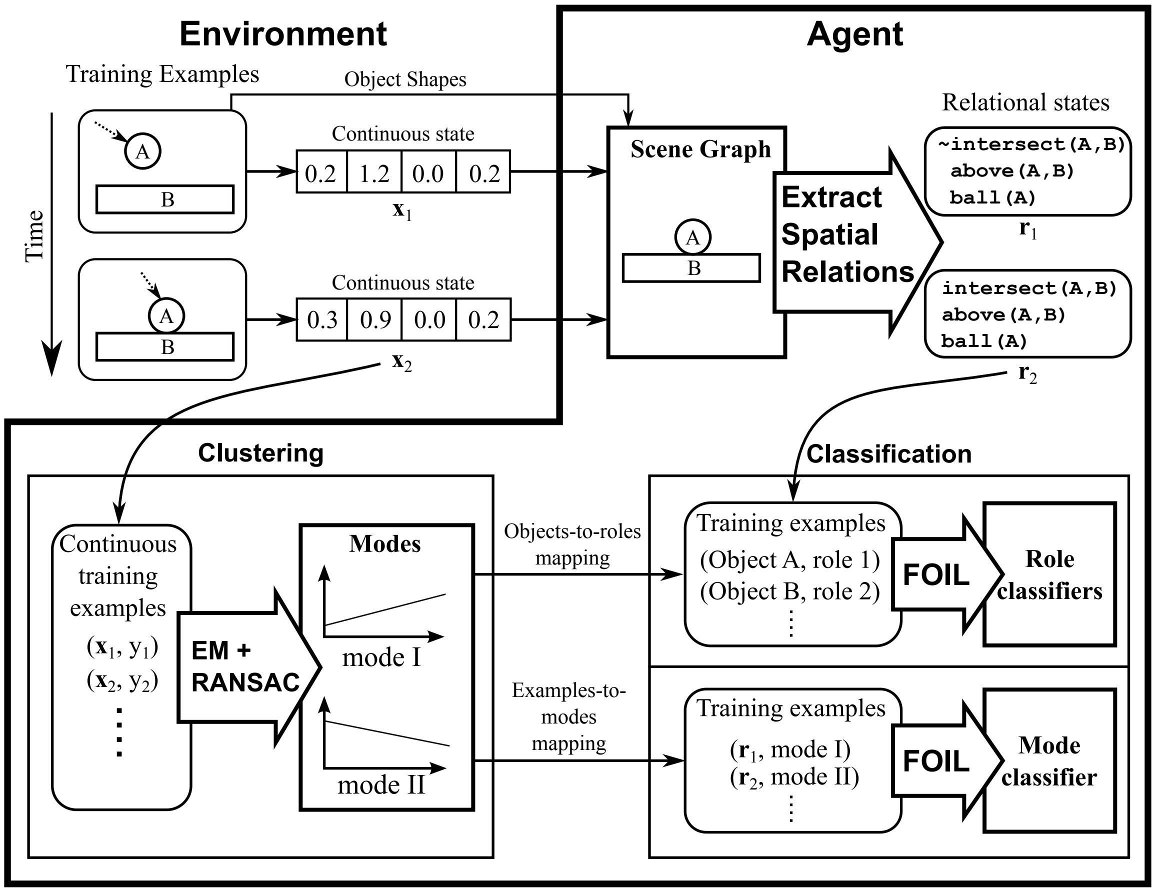


Figure 1.4: Overview of the model learning system.

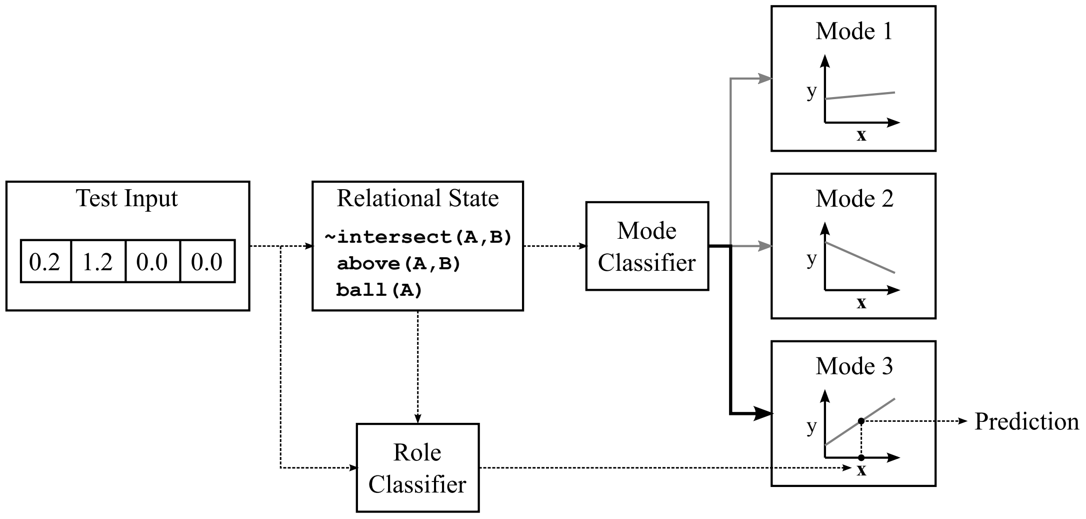


Figure 1.5: Prediction using a multiple modes.

correspond to the modes in the environment, and the linear functions they fit will correspond to the mode functions. We assume that all modes can be modeled with linear functions. The clustering is performed using an Expectation-Maximization-style and standard linear regression. Note that final number of clusters (modes) is not known a priori. New clusters are introduced incrementally when existing clusters cannot accommodate all training data.

Our system variablizes the objects whose properties are tested by the mode functions into roles. It then learns relational rules that assign objects to the correct roles in new situations, which we call *role classifiers*. These relational rules are learned using an inductive logic programming algorithm called FOIL (*Quinlan*, 1990). The rules are first order and assign objects to roles based on their spatial relationships to other objects. Role classifiers allow the learned mode functions to be applied to novel situations where objects have different names than the training examples, and/or the correspondence between training objects and test objects is not categorically apparent.

The clustering problem has a dual classification problem of determining which cluster (mode) a new example belongs to. Our system solves this problem again using FOIL to learn a set of rules that assigns transitions to the correct mode, which we call *mode classifiers*. Mode classifiers are also first order and test spatial relationships between objects. Object names are variablized so that the rules apply to test examples with different objects than the training examples. The rules learned in the mode classifiers can be thought of as preconditions for when a mode will occur.

Prediction using the learned model is straightforward, as shown in figure 1.5. First, the correct mode is identified by feeding the relational state to the mode classifier. The role classifier then assigns objects to the roles in the chosen mode function so that it can be evaluated. An evaluation of the mode function using the appropriate object properties from the test example produces the prediction.

Our approach uses both continuous and relational representations in a mutually beneficial way. Modeling a continuous environment using linear functions is more natural and more accurate than discretizing the state space. It is also more tractable than modeling behavior at the relational level, such as by learning STRIPS operators (see section 2.3). The clustering algorithm essentially solves a large continuous optimization problem and infers hidden labels in the form of mode assignments for training examples and role assignments for objects within each example. These labels are then used as training data for the classification component to learn role classifiers and mode classifiers at the relational level. Learning relational preconditions for modes provides better generalization than using a single mode for all behaviors or distinguishing between modes with continuous classifiers such as SVMs (see section 4.7.2). Learning relational classifiers for roles allows the system to use a mode function learned in one context to make predictions in relationally analogous contexts, such as states with different objects. This kind of generalization is not possible using a purely propositional representation.

Our approach satisfies all the requirements laid out in the previous section. It satisfies **R1** and **R4** by partitioning behaviors into modes and learning mode functions that are only conditioned on the relevant roles in each mode. It satisfies **R2** by learning mode classifiers that are based on spatial relationships instead of absolute coordinates. It satisfies **R3** by explicitly identifying the roles in modes and learning first order classifiers to assign these roles to objects in any context. Finally, our system learns from online training data, so it satisfies **R5**. However, many of our algorithms are not incremental and can occasionally take a few minutes to process an additional training example on a modern processor. This happens when the new example causes major changes to the model's structure, such as discovering a new mode.

CHAPTER II

Related Work

Many methods for action modeling have been proposed, both for relational and continuous environments. Because our system combines both types of learning, our work can be compared to wide variety of approaches. In the following sections, we first compare our approach to continuous model learners and discuss how relational learning and classification allows our approach to learn faster and more generally than purely continuous approaches. We categorize continuous model learners into two broad classes: methods that learn a single function over the entire state space, and methods that partition the state space into regions, and learn submodels over those regions. We then compare our approach to purely relational learners and discuss how modeling continuous effects is simpler than modeling relational effects.

2.1 Single continuous function learners

The simplest approach to modeling a continuous environment is to learn a single function

$$F(\mathbf{x}_t) \rightarrow y_t$$

from training data $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$. Generalization of the training data usually follows a smoothness assumption: test examples will be similar to nearby training examples, where nearness is usually defined as the norm $|\mathbf{x}_{test} - \mathbf{x}_{train}|_k$ for some k. For example, $k = 2$ for Euclidean distance.

Because models in complex environments, such as robotics applications, can be arbitrarily complex, the most popular techniques for model learning are nonparametric. This means that instead of using training examples to adjust a fixed number of parameters, each training example is itself a parameter of the model. Practically, this means that the model can fit any function given enough training data. Therefore, the criterion to judge these methods on is not accuracy at the limit, but how well they can generalize a small amount of training data to new situations.

2.1.1 Locally weighted regression

Locally weighted regression (LWR) (*Atkeson et al.*, 1997a) is an instance-based (*Aha et al.*, 1991) learning algorithm that has been successfully applied in many robot action modeling contexts (*Atkeson et al.*, 1997b). LWR is appealing because it is conceptually simple, non-parametric, and can learn arbitrarily complex functions in the limit. In the learning phase, LWR simply stores all training examples in a large table. To make a prediction for input x, LWR takes these steps:

1. Find the K nearest neighbors of x using some distance metric, typically Euclidean distance.

2. Perform a weighted linear regression on the K neighbors to obtain a linear function f. Each neighbor is weighted based on a function of its distance to the query instance, called the kernel function. A typical weighting function is $K(d) = \mathrm{e}^{-d^2}$.

3. Evaluate $f(x)$ to obtain the prediction.

2.1.2 Gaussian process regression

Gaussian process regression (GPR) (*Rasmussen and Williams*, 2005) is a non-parametric Bayesian inference method. GPR models the training and test data as a multi-dimensional Gaussian distribution, with one dimension per training/test example. Let $\mathbf{X}$ be the matrix of training inputs (one example per row), $\mathbf{y}$ the column vector of training responses, $\mathbf{X}_*$ the matrix of test inputs, and $\mathbf{y}_*$ the test responses. Then

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{y}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(\mathbf{X},\mathbf{X}) & K(\mathbf{X},\mathbf{X}_*) \\ K(\mathbf{X}_*,\mathbf{X}) & K(\mathbf{X}_*,\mathbf{X}_*) \end{bmatrix}\right)$$

$K(\mathbf{A},\mathbf{B})$, is a covariance matrix with each element $K_{ij} = k(\mathbf{A}_i, \mathbf{B}_j)$. k is called the kernel function. A typical kernel function is the "squared exponential":

$$k(\mathbf{a},\mathbf{b}) = \sigma^2 \exp\left(-\frac{(\mathbf{a}-\mathbf{b})^2}{2l^2}\right)$$

σ^2 is the variance of the observation noise, and l is the "characteristic length", which controls the smoothness of the function. The kernel function serves a similar role as the distance metric in LWR. Examples with low covariance affect each other's probabilities less than examples with high covariance. The squared exponential kernel is essentially an exponential on top of the Euclidean distance between examples. GPR predicts the posterior test response (given the training data) to be

$$\mathbf{y}_*|\mathbf{X},\mathbf{y},\mathbf{X}_* = K(\mathbf{X}_*,\mathbf{X})K(\mathbf{X},\mathbf{X})^{-1}\mathbf{y}$$

This expression is called the *conditional distribution* of $\mathbf{y}_*$ given $\mathbf{y}$

GPR is expensive as it requires taking the inverse of $K(\mathbf{X},\mathbf{X})$, which is size $n \times n$, n being the number of training examples. This takes $\mathcal{O}(n^3)$ time, making GPR extremely slow for large training sets. A typical way to deal with this limitation is the take a smaller random subset of the training examples, but this technique may ignore important examples. Recent work has been done on running GPR on individual partitions of the training examples and setting the prediction to a weighted average of the resulting functions (*Nguyen-tuong and Peters*, 2008).

2.1.3 Discussion

The major advantage of these methods is that they are online learners, and thus satisfy requirment **R5** from section 1.3. However, there are several problems with applying these approaches to spatial domains with multiple interacting objects. First, standard LWR implementations use vector norms such as Euclidean distance that weigh all state dimensions equally. Because interactions in spatial domains are often localized, only a small number of state dimensions will matter at any time. There can be cases where examples that agree on relevant dimensions appear more dissimilar than examples that only agree on irrelevant dimensions. This violates requirement **R1**. Extensions to LWR such as Locally Weighted Projection Regression (*Vijayakumar et al.*, 2005) and GPR try to mitigate this effect of irrelevant dimensions.

Another shortcoming of using a vector norm as similarity is that it doesn't capture relationships between dimensions. This is a problem because relative distances between objects often matter more than their individual distances to the coordinate origin. Consider the four training examples and the test example in figure 2.1. Intuitively, example 1 is the most qualitatively similar example to the test instance, so it should be weighed the most when making the prediction. However, in terms of Euclidean distance, example 1 is the most dissimilar example from the test instance. Therefore, methods such as LWR and GPR will weigh example 1 the least in predicting the outcome. This is a violation of requirement **R2**.

The second problem with these methods is a lack of compartmentalization of training examples into individual modes. Basically, real variations in the behaviors

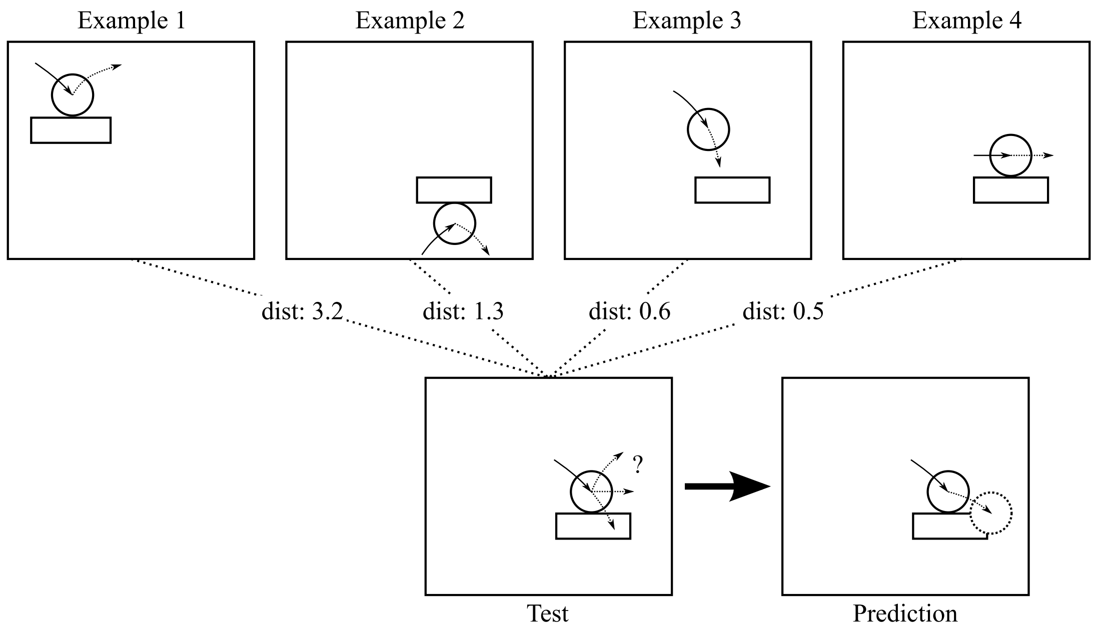


Figure 2.1: Euclidean distance doesn't correlate with behavioral similarity.

of similar examples due to being in different modes is confounded with variation due to noise. Whereas "averaging out" variations due to noise can lead to more accurate predictions, averaging out real variations leads to predictions that are not accurate for any mode. This is shown in the prediction panel of figure 2.1. Training example 2 is from the mode of the ball bouncing against the bottom of a platform, example 3 is from the falling mode, and example 4 is from the rolling mode. Because they all contribute to the final prediction, the prediction becomes an average of several modes and is not accurate for any mode. This is a violation of requirement **R4**.

Another problem is that these methods use propositional representations, so there is no way to apply a model learned on one set of objects to another set of objects, which violates requirement **R3**. In propositional representations, the state is assumed to have fixed dimensionality and each attribute is assumed to be categorically unique (e.g. age versus height). Attributes across different training examples are assumed to have a single obvious correspondence (weight corresponds with weight, height corresponds with height). Both these assumptions are violated when the state contains many objects of the same type, such as in the stairs example in section 1.2.5.

2.2 Input space partitioning continuous learners

The following methods are similar to ours in that they learn a set of relatively simple submodels that together cover the entire space of possible behaviors. In other words, let $\mathcal{R}^d$ be the input space and $\mathcal{R}$ the response space, so that the model can be described as $F : \mathcal{R}^d \to \mathcal{R}$. These methods learn a set of submodels

$$\left\{f_i : \mathcal{R}^d \to \mathcal{R} \,\mathrm{for}\, i \in 1, \ldots, n\right\}$$

and a partitioning of the input space

$$\left\{p_i \subseteq \mathcal{R}^d \,\mathrm{for}\, i \in 1, \ldots, n \text{ s.t. } p_1 \cup \ldots \cup p_n = \mathcal{R}^d \wedge \forall_{i,j} p_i \cap p_j = \emptyset\right\}$$

Each function f_i is trained only on the examples that are contained in the paired partition p_i. Evaluating F on input $\mathbf{x}$ is just a matter of finding the partition that contains $\mathbf{x}$ and evaluating the associated submodel:

$$F(\mathbf{x}) = f_i(\mathbf{x}) \text{ where } \mathbf{x} \in p_i$$

2.2.1 Model trees

Model trees (*Quinlan*, 1992) are decision trees whose interior nodes contain a simple test that partitions of the input space in two and whose leaves are functions. Prediction using a model tree involves routing the test input from the root of the tree to a leaf, following the branch at each interior node based on the test result, then evaluating the function at the leaf. Model trees are "grown" by choosing a test to split the training examples, then recursively growing the left and right subtrees using the respective partitions. Growth is terminated when the examples at a leaf can be satisfactorily modeled by a single function, which is usually linear. The choice of the test at each interior node is based on the *splitting rule*. For example, the splitting rule of the well known M5 algorithm (*Quinlan*, 1992) chooses a partitioning that minimizes the standard deviation of the training response variables. The test itself usually partitions the input space by comparing a single dimension with a single value, such as $\mathbf{x}_4 > 12.0$. This results in the points on each side of an axis-aligned hyperplane being placed in one partition.

2.2.2 Mixture of sigmoids approach

Toussaint and Vijayakumar (2005) developed an algorithm that partitions training data into a collection of linear local models using Expectation Maximization and assigns test input to local models using a product-of-sigmoids classifier. The product-of-sigmoids classifier essentially partitions the input space into a set of polyhedrons. Each face of the polyhedron is established by a single sigmoidal function, a binary classifier that chooses between the more likely of two possible local models for the input. Taking the product of all sigmoids is like performing an *and* on all the binary classifiers. All test examples whose inputs fall into a particular polyhedron are predicted using the local model associated with that polyhedron.

This approach was the starting point of our work. Even though the product-of-sigmoids classifier is more expressive than the axis-aligned hyperplane partitions used in model trees, they still exhibit weak generalization in domains where spatial relationships are the natural partitions.

2.2.3 Multi-Modal Symbolic Regression

Multi-Modal Symbolic Regression (MMSR) (*Ly and Lipson*, 2012) is an algorithm that learns models of hybrid dynamical systems. Hybrid dynamical systems are finite automata whose states are analogous to our concept of modes. The system is always in exactly one state and will exhibit the behavior associated with that state. The system may transition to another state when certain conditions are met, as governed by the transition function.

Like our algorithm, MMSR uses Expectation Maximization to assign training examples to modes. MMSR models the behaviors of individual states using an algorithm called Eureqa (*Schmidt and Lipson*, 2010). Eureqa is essentially a nonlinear regression algorithm that uses genetic programming to search the space of symbolic equations for one that best fits the data. The ability to model complex functions increases the potential for a single mode to overfit the data and cover more training examples than it should. MMSR uses the Akaike Information Criterion (*Akaike*, 1974) to balance equation complexity with goodness of fit. MMSR also learns a model of the system's transition function. It uses a similar evolutionary technique adapted for classification to find equations that describe conditions under which state transitions occur. Because these equations are potentially nonlinear, the state space partitioning learned by MMSR is more expressive than the hyperplane partitioning schemes used in the previously discussed works.

MMSR's partitioning scheme is more expressive than using combinations of hyperplanes, but it still operates in absolute coordinates, and is therefore not as natural to adapt to the domains we are interested in as spatial relations. MMSR's ability to model nonlinear mode functions is something our system is unable to do. We avoided nonlinear regression mainly because of the problems with EM and overfitting that is described in the MMSR work. Exploring how to use the AIC to prevent overfitting in our system is left as future work.

2.2.4 Discussion

All of the above methods partition examples along relevant state dimensions and learn functions that are only conditioned on relevant dimensions, so they satisfy requirement **R1**. They all use propositional representations, so they fail requirement

R3.

Predictions made by models learned with these methods are only affected by the training examples that share a partition with the test input. This is different from the single model learning algorithms described previously, in which every training example influences every prediction, albeit the influence is weighted by a similarity metric. Therefore, these methods get closer to modeling independent behavioral modes and the possibility for abrupt changes to occur when transitioning between modes, satisfying requirement **R4**.

Model trees and product-of-sigmoids models partition the input space using hyperplanes located at absolute coordinates. This means that a model learned in one part of the space cannot be applied to another part of the space. Hence these methods fail requirement **R2**. Furthermore, distinct modes due to external influences are usually based on spatial relationships between objects, and many spatial relationships are hard to capture with hyperplanes located at absolute coordinates. For example, whether two objects intersect is conditioned on their position vectors being close to each other, not their absolute values. Therefore, hyperplane-based partitions are not expressive enough to describe the entire space where a particular submodel applies, and generalization suffers as a result. On the other hand, MMSR is capable of partitioning the space using nonlinear functions, so it does not suffer from these shortcomings.

The product-of-sigmoids learner is online and satisfies requirement **R5**, and there is an online and incremental version of model trees (*Potts*, 2005), but MMSR is not online. Furthermore, MMSR has reported run times of 10 to 40 minutes for toy problems on standard modern hardware (*Ly and Lipson*, 2012), which is orders of magnitude slower than either of the other approaches. This is not surprising considering the high expressivity of symbolic regression and use of evolutionary algorithms.

2.3 Relational action model learning

Much of the early work in action modeling was on learning STRIPS operators in relational domains (*Wang*, 1995; *Carbonell and Gil*, 1996). The work presented in this thesis grew out of previous systems we built that performed instance-based relational model learning (*Xu and Laird*, 2010) and combined relational model learning with LWR (*Xu and Laird*, 2011).

The relational aspects of our system (mode and role classifier learning) most closely relates to the work of *Kaelbling et al.* (2007). They built a system that learns STRIPS-style "Noisy Deictic Rules" (NDR) of relational domains from noisy training examples. NDRs are first-order and contain "deictic references," references to objects that affect an action's outcome or are modified by the action, but are not in the action parameters. In blocks world for example, the $\textit{pick-up}(X)$ operator has the parameter X for the block to pick up. However, one of its effects includes deleting the relation $\textit{on}(X, Y)$, where Y is the block or table that X was originally on. Therefore, Y is a deictic reference in $\textit{pick-up}(X)$. Deictic references correspond to roles in our models. Each deictic reference has a conjunction of first-order literals that restrict which objects it can bind to, which correspond to role classifiers in our system. For the deictic reference Y in the *pick-up* operator, the restriction is that $\textit{on}(X, Y)$ is true.

Even though they fulfill the same function, deictic references and roles are learned in completely different ways. Deictic references are discovered as part of a general search through the space of possible rules. This search involves iteratively applying a set of search operators to modify a seed NDR in ways such as adding or removing literals to/from a rule's precondition or postcondition, adding or removing deictic refences, adding or removing literals to deictic reference restrictions, etc. The search is greedy and maximizes the log-likelihood of the rule set with a complexity penalty. The discovery of deictic references is therefore folded into the process of maximizing this objective function along with learning the other aspects of the NDR, such as its preconditions or its effects.

In our system, roles are identified during clustering, separate from and strictly before learning role classifiers (which correspond to the descriptions of deictic references) and mode classifiers (which correspond to rule preconditions). When our system reaches the stage to learn role classifiers, it already a set of roles and labeled examples of objects that are and are not suitable for each role. Therefore, learning role classifiers is a straightforward supervised learning problem. This makes our system both simpler and more powerful, in that it is able to learn complex role classifiers made of conjunctions and disjunctions of many literals. The NDR learner is unable to learn complex conjunctions for its deictic reference restrictions because its greedy search requires each additional literal to increase the log-likelihood of the rule more than it increases the complexity penalty. This is not possible if a conjunction of literals only increases rule correctness when it is complete, which is often the case in domains like blocks world. The NDR learner compensates for this by introducing another level of search through the space of possible conjunctions of basic literals. This search tries to combine basic relations into useful conjunctions called concepts, and then run the NDR learning procedure using a vocabulary of both basic relations and concepts. If the resulting rules have higher objective scores, the concept is considered useful and kept in the system. Although the authors do not report the computation time required for this search, we suspect that it is expensive.

Our system is simpler because each model it learns tracks the continuous change of a single property of a single object, whereas NDRs and all STRIPS-style rules model relational changes that can involve arbitrarily many objects for a single action. *Huffman and Laird* (1992) discuss the advantage of modeling action effects using non-relational, concrete representations. They point out that a continuous, propositional state representation implicitly encodes the combinatorially large number of relationships that are explicitly represented in the equivalent relational state. An action model of the continuous propositional state space therefore only has to update a bounded number of properties, while an action model of the relational state space must update an exponential number of relationships in the worst case. This problem with relational action models is known as the *ramification problem* *Ginsberg and Smith* (1988). For example, consider the situation shown in figure 2.2. Box A undergoes a simple linear translation between states 1 and 2, which can easily be modeled in the continuous propositional state space. However, this results in a complex set of changes in the relational state space. The number of literals that can change in this situation can be arbitrarily large depending on the number of objects in the environment, making a general rule that describes this transition difficult to learn. By modeling only the continuous effects of single object properties, our system avoids the ramification problem. Furthermore, these models can still be used to predict the relational effects of actions by using the simulate-then-derive strategy introduced by *Huffman and Laird* (1992). This strategy is to simulate an action in the continuous state space, obtain the final continuous state, and then calculate the new relational state by testing the truth value of every relation. Although simulating the entire state requires our system to learn one model for each property of each *type* of object, the total number of models is linear with the number of object types in the environment, which is usually much smaller than the number of objects. On the other hand, the number of effects to be modeled by a single rule of a relational model is combinatorial with the number of object instances in the worst case.

The unlimited generality of a relational rule also makes action modeling intractable in another way: the learner must search the space of trade offs between a large number of specific rules and a small number of general rules. This trade off is also present in our system, where single modes can use arbitrarily complex non-linear mode functions to model a large number of behaviors. Our solution is to fix the trade off a priori and

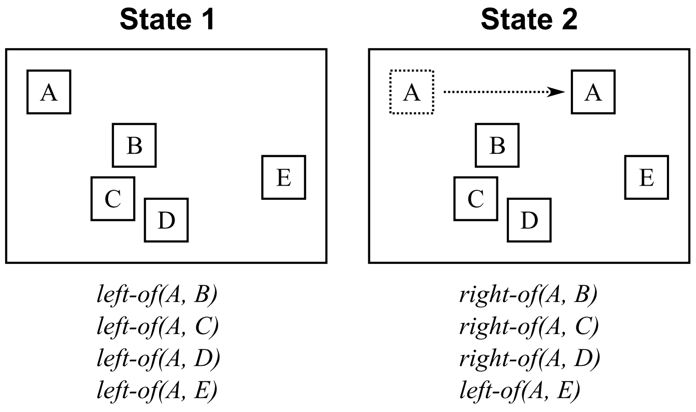


Figure 2.2: Example of how a simple linear change in the continuous state space results in complex changes in the relational state space.

restrict mode functions to be linear. This solution is not suitable if the environment has non-linear modes that cannot be decomposed into simpler linear modes.

2.4 Object-Oriented Markov Decision Processes

Diuk et al. (2008) describe a system that learns models in object-oriented domains with continuous attributes called Object Oriented Markov Decision Processes (OOMDP). In this formulation, the environment is composed of a set of objects, each belonging to a class. The class of the object defines what attributes it has and how it behaves. Relations are boolean functions on two objects that test conditions about the objects' attributes, such as whether the objects touch. Actions are modeled as a set of condition-effect pairs, such that when the agent performs an action, any effects whose associated conditions are met will take place. Conditions are conjunctions of tests for the truth values of relations. Effects can either set an object's attribute to an absolute value, or increment or decrement the attribute value by a particular amount.

The OOMDP representation has many similarities to our representation. Both define objects to have types (classes) that determine the attributes they have and their behaviors. Both define relations on multiple objects whose truth values are based on the attributes of the objects. Both allow for qualitatively different effects for one action based on preconditions that test the truth values of relations.

Our approach is more general than OOMDPs in many ways. The preconditions of effects in OOMDPs are propositional even though they test relations. They cannot represent concepts such as $\textit{on-top}(A, B) \wedge \textit{on-top}(B, C)$ and enforce that the variable B is bound to the same object in both conjuncts. Our mode classifiers are first order and can represent such concepts. While models in OOMDPs can only represent effects as setting object attributes to specific values or incrementing or decrementing by a fixed amount, our models learn effects that are general linear functions of object attributes. Furthermore, because OOMDP effects are not conditioned on attributes of other objects, their models don't address the role assignment problem which is unavoidable in more flexible representations of effects. Our approach solves the role assignment problem by learning role classifiers using FOIL. The increased generality of our approach comes at the cost of higher computational complexity.

CHAPTER III

System description

In this chapter, we present a detailed description of our model learning algorithm. We will first describe an experimental domain which we will use as a running example throughout the chapter. Then we will describe each component of our system as shown in figure 1.4.

3.1 The box pushing domain

The demonstration domain we will use consists of a truck and a stack of boxes, as shown in figure 3.1. The truck moves toward the right and generates a constant force that pushes the stack of boxes. The left side of the figure illustrates the forces on the

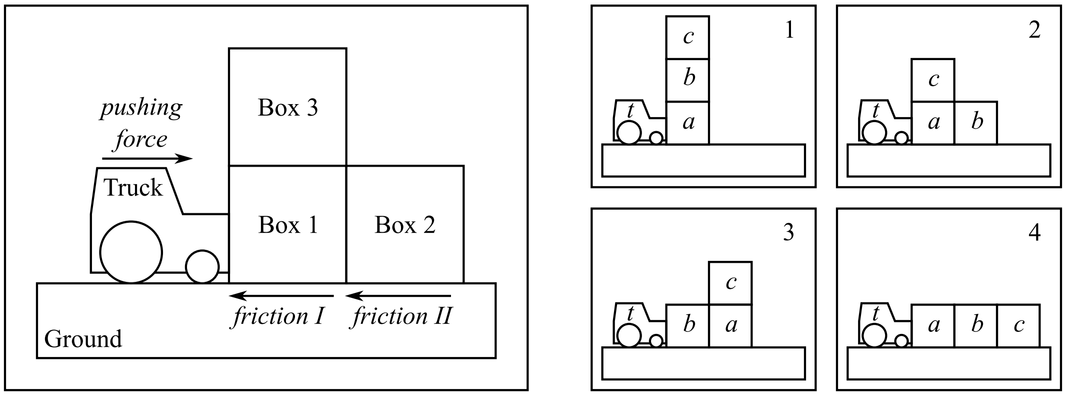


Figure 3.1: The box pushing domain (left) and the possible configurations of the boxes (right).

system. Each box has the same mass and a different non-zero friction constant. The frictional force generated against the ground opposes the pushing force. Each vertical stack of boxes generates a different frictional force. We use the basic Coulumb model of friction, so the friction force is calculated as the product of the downward normal force generated by the stack's weight and the friction constant of the bottom-most box. The total force on the system determines the acceleration of all the objects and in turn the velocities of the objects. We set up our system to learn a model of the truck's horizontal velocity.

The right side of figure 3.1 shows the possible configurations of the boxes. Note that the labels a, b, and c are labels for positions and not the names of the boxes. So each configuration in the figure corresponds to six concrete states because the identities of the boxes can be permuted amongst the positions. Since configurations 2 and 3 have identical stacks of boxes except for their horizontal order (which is irrelevant in determining behavior), we expect them to behave identically. However, configurations 1, 2, and 4 distribute boxes among stacks differently, so we expect different frictional forces in each and thus different behaviors. These different behaviors will be modeled using different modes. Furthermore, the model will have to learn a mode classifier that can distinguish the different modes based on the configurations of the boxes.

We train the model on a sequence of 48 blocks. Each training block begins in one of the 4 configurations, one of the 6 permutations of box identities, and one of 2 random seeds. The random seed affects the size of the boxes and their positions in absolute coordinates. This shows that our learned model generalizes across these random variations, and also prevents it from overfitting some accidental regularities that we didn't anticipate. 40 training examples are generated from each initial state by simulating the system for 40 time steps. This results in a total of 1920 training examples. This is actually more training examples than needed for the system to learn the correct model, but we want to show that the model can correctly unify a diverse training set under a compact description.

3.2 Clustering

The clustering component is responsible for discovering new modes, learning the linear function for each mode, identifying the roles in each mode, and assigning each point in the training data to a mode. Input into the clustering component is a stream of continuous training data

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$$

where $\mathbf{x}_i$ is the initial state vector, and y_i is the value of the target dimension in the post state, as discussed in section 1.2. The outputs of clustering are

- A set of modes. Our system starts out with only a single "noise" mode that serves as a catch-all for examples that don't belong to any other modes. As the system accumulates examples, it tries to identify subsets of examples that are highly correlated and create new modes to accomodate them. This is accomplished with the RANSAC algorithm.

- A linear function that describes the behavior of each mode. Since the behavior of each mode is distinct, our system learns a function for each mode, called the *mode function*. It does this by fitting a line to the examples assigned to the mode using linear regression. We assume that all behaviors can be described by linear functions.

- A mapping from training examples to modes. We assume that each training example was generated by a unique mode, but the true mode is not observed. The clustering algorithm tries to deduce the mode responsible for each training example based on how well it fits the linear function of each mode. More precisely, the mode assigned to example (x, y) is $\arg\min_{m \in M} |y - F_m(x)|$ where M is the set of modes and F_m is the function for mode m.

Note that the second and third items are interdependent: the mode function is learned by fitting a line to the examples assigned to it, and examples are assigned to mode based on how well they fit the mode functions. We solve for both sets of parameters (parameters of mode functions and assignments of examples) simultaneously using Expectation Maximization (EM), discussed below.

The use of the terms "input" and "output" may be misleading here. Clustering does not start from scratch with each new input, and does not return a complete copy of its output with each run. Instead, new inputs trigger an iterative process that runs on persistent structures until convergence. Output simply means that the process has converged and the structures can be inspected. Therefore, clustering can be considered an incremental algorithm in form, although it is not computationally incremental and may inspect the entire history of inputs during some iterations.

Figure 3.2 shows the clustering process. When a new training example arrives, EM is first run until all mode assignments and mode functions converge. Next, if the noise mode contains enough unassigned examples, RANSAC is run to look for a new mode. If a new mode is not found, then the system has converged and waits for the next input. Otherwise, the system tries to unify the new mode with existing modes. Regardless of whether any unification was successful, EM is run again to find the best mode assignments and functions in light of the new mode. We will now describe each algorithm in the flowchart in detail.

3.2.1 Learning mode functions

Mode functions are linear and have the form $F_m(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$, where $\mathbf{w}$ is a vector of weights $(w_1, w_2, \ldots, w_k)$ and $\mathbf{w} \cdot \mathbf{x}$ is the inner product of the weights with the

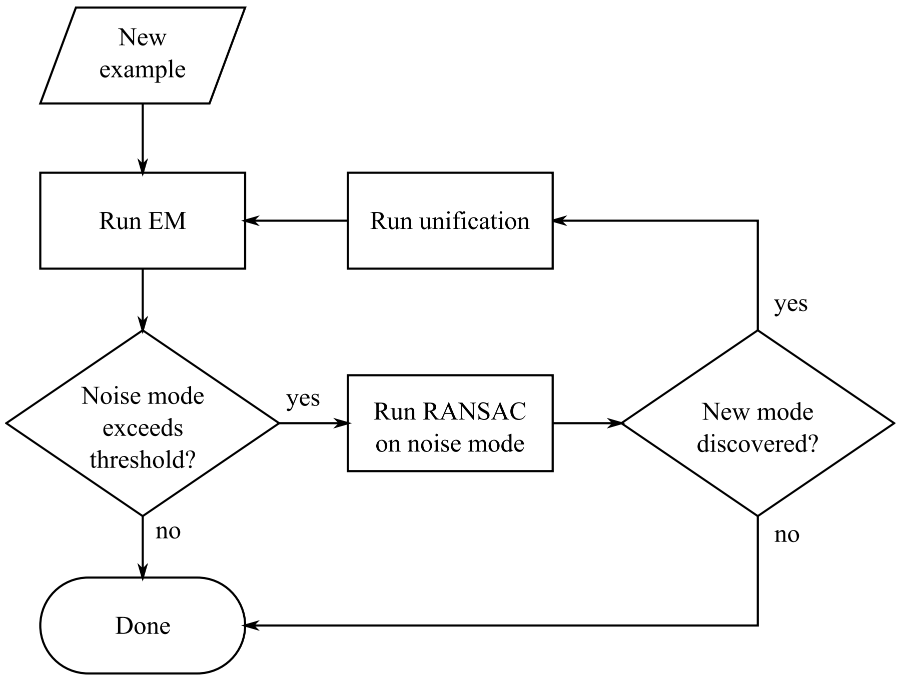


Figure 3.2: Flowchart for the clustering process.

state vector. Learning the mode function involves finding the best set of weights by applying linear regression to all training examples assigned to the mode.

Linear regression is a well-understood problem. Given a set of training examples

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$$

the basic linear regression algorithm finds a set of weights that minimizes the residual training error

$$\arg\min_{\mathbf{w}} \left\{ \sum_{i=1}^{n} (y_i - \mathbf{w} \cdot \mathbf{x}_i)^2 \right\}$$

Notice that if an element in $\mathbf{w}$ is zero, then the corresponding element in $\mathbf{x}$ would have no impact on the linear function's value because it's zeroed out when multiplied by its weight. Typically, we want linear functions to have as few non-zero weights as possible, because functions that depend on fewer explanatory variables tend to be more general according to Occam's Razor. For example, suppose that two dimensions of the state vector have a constant value of 1 in all training examples. Clearly, these dimensions do not play a role in determining the function's output. However, a naive linear regression algorithm may set the weights for these dimensions to 10^6 and -10^6 without incurring additional residual error, since they cancel each other out. But when applied to test examples where the dimensions in question vary even slightly away from value 1, the learned function will incur extreme errors. On the other hand, if the weights were set to 0, the function would not be affected no matter how much the values in the test examples vary.

The practice of minimizing the number of nonzero weights is called *regularization*. Typically, regularization techniques supplement the residual error with a slight penalty for each non-zero weight. For example, the popular ridge regression technique minimizes the residual error plus the l^2 norm of the weight vector:

$$\arg\min_{\mathbf{w}} \left\{ \sum_{i=1}^{n} (y_i - \mathbf{w} \cdot \mathbf{x}_i)^2 + \lambda |\mathbf{w}| \right\}$$

The λ constant trades off the aggressiveness of the regularization with residual error. Some other regularization techniques include LASSO and forward-stepwise regression (*Hastie et al.*, 2002).

Regularization is especially important in our model learning system. The state vector $\mathbf{x}$ contains at least nine dimensions for each object in the environment (the basic position, rotation, and scaling transforms), so even with very few objects, the dimensionality of the vector is high. But for most modes, only a small number of dimensions actually affect behavior. For the reasons discussed above and in section 3.2.3, it is important that only the truly relevant dimensions have non-zero weights.

Our system uses forward-stepwise regression for linear regression and regularization. This technique has empirically proven to be more robust in our experiments than ridge regression and LASSO. The basic idea of forward-stepwise regression is to repeatedly run basic linear regression on subsets of the state vector dimensions, each time adding only one dimension that results in the best improvement in residual error, until a threshold is reached. Pseudo-code for this technique is shown in algorithm 1. Note that the pseudo-code illustrates a naive and inefficient implementation of the algorithm. The actual implementation in our system uses the sweep operator (*Lange*, 2010) to perform multiple regressions efficiently.

Algorithm 1 Forward-stepwise regression

```
1: procedure FORWARD-STEPWISE-REGRESSION($X, Y$)
2:     Let $X$ be a $n \times m$ matrix, with each row being a training example
3:     Let $Y$ be a $n \times 1$ vector of the corresponding response values
4:
5:     $c \leftarrow \emptyset$                                  ▷ $c$ is the set of chosen regressors
6:     $r \leftarrow 1 : m$                                ▷ $r$ is the set of remaining regressors
7:     $w \leftarrow \text{LinearRegression}(X, Y)$
8:     $S \leftarrow \frac{1}{n}\sum_{i=1}^{n}(Y_i - w \cdot X_i)^2$                 ▷ $S$ is MSE with all regressors
9:     $C_{best} \leftarrow \inf$
10:    while $r \neq \emptyset$ do
11:        $d_{best} \leftarrow \emptyset$
12:        for $d \in r$ do
13:            $c' \leftarrow c \cup d$
14:            $X' \leftarrow [X_i \ \forall i \in c']$                 ▷ $X'$ is matrix with $c'$ columns of $X$
15:            $w \leftarrow \text{LinearRegression}(X', Y)$
16:            $SSE \leftarrow \sum_{i=1}^{n}(Y_i - w \cdot X_i')^2$          ▷ $SSE$ is residual error with $c'$
17:            $C_p \leftarrow \frac{SSE}{S^2} - n + 2|c'|$                   ▷ Mallow's $C_p$ statistic
18:            if $C_p < C_{best}$ then
19:                $C_{best} \leftarrow C_p$
20:                $d_{best} \leftarrow d$
21:            end if
22:        end for
23:        if $d_{best} \neq \emptyset$ then
24:            $c \leftarrow c \cup d_{best}$
25:        else
26:            return $c$
27:        end if
28:    end while
29: end procedure
```

3.2.2 New mode discovery

Our model learner does not have a priori knowledge about which modes exist in the environment, and must discover these modes from training data using the approach of *Toussaint and Vijayakumar* (2005). The model learner begins with a single noise mode. All new training examples that do not fit any existing modes are assigned to the noise mode. Therefore, training examples belonging to undiscovered modes will be in this set. The problem is that these examples will be mixed in with examples from other undiscovered modes and genuine noise, so they must be separated out. We assume that it is unlikely for a large number of random examples or examples belonging to different modes to fit the same linear function. So if a large set of examples in the noise mode are found to fit the same line, we assume they belong to the same undiscovered mode, and create a new mode with those examples. Our system uses Random Sampling and Consensus (*Fischler and Bolles*, 1981) (RANSAC) to distill new modes out of the noise mode.

RANSAC is an algorithm for performing linear regression on data sets with large percentages of outliers. The goal is to fit a line to a set of "inliers" without letting the outliers influence the final fit. In the context of our problem, the inliers are examples from the same undiscovered mode, and the outliers are truly noisy data or examples from other undiscovered modes. The basic idea of RANSAC is to repeatedly draw small samples from the data set in hopes of getting one without outliers, fit a line to the sample, then find all other points that also fit the line. Assuming the data set D contains (x_i, y_i) pairs, RANSAC follows this procedure:

1. Draw a random sample $S \subset D$.

2. Fit a linear function f to S using the linear regression method described in section 3.2.1.

3. Find all other points $(x, y) \in D$ such that $|y - f(x)| < 2\sigma$. Call this set S'.

4. If $|S'| > T$, then return f and S'. Otherwise repeat from 1 unless the maximum number of iterations N is exceeded.

T, σ, and N are all free parameters in our system. T is a threshold parameter that is manually set in our system. Because RANSAC is used to discover new modes in our system, the value of T directly impacts the quality of the learned model. If T is set too low, the system may learn spurious modes that overfit accidental linear relationships between examples. If T is set too high, then more data than necessary is required for the agent to discover new modes. Furthermore, in online training contexts, there is no upper limit that will guarantee that the system has a good distribution of examples before committing to a new mode. The value to use trades off speed of learning with the goodness of the learned models. We suspect that erring on the side of slower learning will be better than learning overspecific models in most situations. T is set to 40 in most of the experiments described in this thesis.

We assume that all training examples can be corrupted by Gaussian noise with mean 0 and standard deviation σ. This free parameter is used throughout the system as the expected standard deviation resulting from noise. Here, we set the maximum tolerance for a point to fit a line at two standard deviations away. Setting σ too high will result in the system folding distinct behaviors into a single mode. Setting σ much lower than the actual environment noise will result in the system being too selective and not learning any new modes at all. We empirically demonstrate these effects in section 4.6.

N is the maximum number of iterations to RANSAC. Higher values of N cause the system to waste time searching for new modes in the noise data that might not exist. Lower values of N cause the system to miss new modes. We set N to 2000 in our experiments because that value empirically worked. An adaptive policy that decreases N over time may result in faster performance by reducing the amount of time spent looking for non-existent modes.

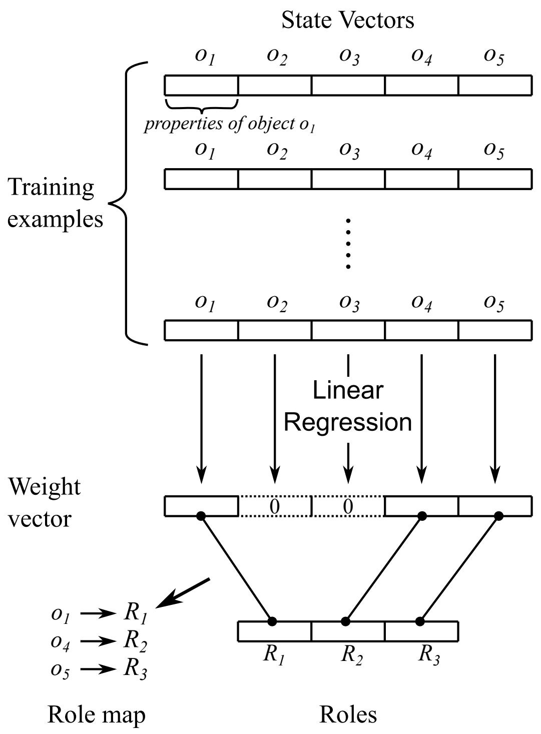


Figure 3.3: Identifying mode roles through linear regression

3.2.3 Identifying roles in mode functions

Recall from section 1.2.5 that every mode has a set of roles that define which objects are involved in the mode and how their properties affect behavior. We define a role to be a set of non-zero weights in a linear mode function that test properties of the same object. For example, the mode function for an elastic collision between two balls is

$$v'_A = k_1 v_A + k_2 v_B$$

where $k_1 = \frac{m_A - m_B}{m_A + m_B}$ and $k_2 = \frac{2m_B}{m_A + m_B}$. Assuming the masses m_A and m_B are constants, this function has two roles. The first role is that of the ball with mass m_A, with weight k_1 for its velocity, and the second role is the ball with mass m_B, with weight k_2 for its velocity.

When a new mode is discovered via RANSAC, its roles are implicitly identified through the linear regression process. This is illustrated in figure 3.3. Each state vector in the training data is composed of a sequence of property vectors, with each property vector enumerating the properties of a single object. The weight vector learned by performing linear regression on a set of state vectors can be segmented in the same way. Each segment with at least one non-zero weight corresponds to a role in the mode. Segments with all zero-valued weights are discarded. The linear regression produces two results: a set of roles and weight vectors for those roles, and a mapping from the objects in the training data to those roles. The latter is used when learning role classifiers, as will be discussed in section 3.5.

This approach requires that the objects that fulfill the same roles in each training example used in RANSAC must occupy the same positions in the state vectors. In other words, they must be aligned in the same "columns" in figure 3.3. This is a reasonable requirement given that our training examples are obtained online and thus come from continuous traces of the environment. It's likely that the same objects will fulfill the same roles over an extended period of time. Furthermore, this requirement only applies to the discovery of new modes. After the mode and its roles are established, our system can fit additional examples with different role assignments to the mode by searching over all possible assignments for the best fit (see the next section). This kind of search is not possible when the roles have not been fixed, since it would require trying to discover a linear relationship over all possible role permutations of every training example, which would be prohibitively slow.

3.2.4 Fitting examples to existing modes

In the previous section, we described how RANSAC is used to discover new modes. Here, we describe how our system assigns new training examples to the set of discovered modes, and how the mode functions are continuously updated to fit new training examples. This is accomplished using Expectation maximization (EM) (*Hastie et al.*, 2002). EM is an algorithm for simultaneously solving for two sets of interdependent unknowns. In the context of our model learning algorithm, these unknowns are

1. The parameters of the linear functions for each mode.

2. The mode that generated each data point.

The two sets of unknowns are interdependent because the parameters of a mode directly determine which data points are plausibly generated by the mode. The basic idea of EM is to iterate between two steps:

- E-step. Assuming the mode parameters are correct, calculate $P(\mathbf{x}_t, y_t | m_t = m)$, the probability that example $(\mathbf{x}_t, y_t)$ was generated by mode m, for all time steps t observed so far. Also calculate the mode that most likely generated each example ($m_t = \arg\max_{m \in M} P(\mathbf{x}_t, y_t | m_t = m)$).

- M-step. Assuming the probabilities from the E-step are correct, use linear regression to fit each mode's function F_m to the set of points that were most likely generated by it $\{(\mathbf{x}_t, y_t) | m_t = m\}$.

This process iterates until convergence or a maximum iteration count is reached.

The probability that the data point $(\mathbf{x}_t, y_t)$ belongs to mode m is based on the residual error ϵ between the observed target value y_t and the value predicted by the mode function. Due to the multiple possible assignments of the objects at time t to the roles in F_m, ϵ is calculated as the minimum possible error over all possible assignments:

$$\epsilon = \arg\min_{\mathcal{R}} |y_t - F_m^{\mathcal{R}}(\mathbf{x}_t)|$$

where $F_m^{\mathcal{R}}(\mathbf{x}_t)$ is the mode function evaluated on $\mathbf{x}_t$ using role assignment $\mathcal{R}$. Because we assume Gaussian noise with variance σ^2 in the training data, the probability of observing residual error ϵ given that example t belongs to mode m follows a normal distribution with mean 0 and variance σ^2:

$$P(\mathbf{x}_t, y_t | m_t = m) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-\epsilon^2}{2\sigma^2}\right\}$$

Since this calculation requires a search over all possible role assignments, it could be expensive if there are many objects of the same type in the environment. But we didn't encounter this problem in our experiments.

It is difficult to characterize the complexity of EM, but the use of linear regression in the M step suggests it is at least $O(n^3)$. We use several strategies to cut down on the actual runtime of EM. In the E step, our algorithm persistently caches all calculated probabilities and only updates the values for modes whose functions have changed. In the M step, our algorithm tries to avoid refitting functions as much as possible. When new examples are added to a mode (meaning $m_t \neq m$ in the previous EM iteration and $m_t = m$ now), a linear regression is not performed if the existing function fits the new examples well enough. Removal of examples from a mode does not initiate a linear regression if the function fits the remaining examples well enough. By "well enough," we mean the maximum error $\max_{\mathbf{x}_t}(F_m^{\mathcal{R}}(\mathbf{x}_t) - y_t) < T$, where T is a free threshold parameter. In practice, we found that most mode functions converged quickly and did not require refitting thereafter.

3.2.5 Mode unification

Our learning algorithm runs online and learns from available data at any point in time. In some situations, the system will learn a mode that fits the observations up to one point in time at the expense of learning a more general mode that fits future observations as well. For example, suppose that an agent receives training examples of a ball rolling at a constant speed in the positive x direction. After a sufficient number of training examples, it may learn a mode for the ball's x-velocity with the function $vx_{t+1} = c_1$, where c_1 is some constant. Suppose that later on, the agent gets training examples of the ball rolling in the negative x direction and learns a new mode function $vx_{t+1} = c_2$. However, a more general mode that would cover rolling in both the positive and negative directions is $vx_{t+1} = vx_t$, i.e. the ball maintains its x velocity.

The problem here is that the model is being learned online, so contiguous training examples tend to come from similar behaviors rather than a uniform sampling of the behavior space. In this situation, learning the overspecific mode first is the right thing to do, since there is no upper bound on the number of training examples to wait for before a sufficient diversity is achieved. Furthermore, the agent may be able to use the overspecific modes to make accurate predictions, such as if the ball kept rolling.

The best thing to do, then, is to learn the overspecific modes as soon as possible, utilize them when possible, but then learn a more general mode when additional training examples become available. Our system tries to do this by *unifying* newly learned modes with existing modes, meaning it tries to find a function that fits the examples from both the existing and new modes. If it is successful, the agent will discard both overspecific modes and keep the generalized mode.

In the case of the rolling balls, upon discovering the negative rolling mode, the system will first try to unify it with the positive rolling mode, and find the more general mode with function $vx_{t+1} = vx_t$. Both the negative and positive rolling modes will then be discarded in favor of the general mode.

The unification algorithm runs each time a new mode is discovered. For each existing mode, it uses RANSAC to search for a function that covers the examples in both the existing mode and the new mode. If RANSAC discovers a function that covers a large ratio R_{unify} of the examples, that function is used to create a new mode. Both the existing mode and the overspecific new mode are then discarded. In our experiments, R_{unify} is set to 0.9.

3.2.6 Modes learned in the box pushing domain

We now analyze the modes learned in the box pushing domain to make the previous algorithm descriptions more concrete. First we analytically derive the forms that the modes should have, then show the modes actually learned.

The acceleration of the truck can be calculated using the equation $F = Ma$. Since all objects in the environment are moving together, we treat them as a single point mass, so $M = m_{truck} + 3m_{box}$, where m_{truck} is the mass of the truck and m_{box} is the mass of a single box. Furthermore, the total force on the system is a combination of the pushing force from the truck and the frictional forces from the boxes sliding on the ground:

$$F = F_{truck} - F_{friction}$$

The frictional force can be further decomposed into the friction experienced by each stack, which depends on the configuration of the boxes. In configuration 1 (see figure 3.1, the frictional force is

$$F_{friction} = C_a \times 3Gm_{box}$$

where C_a is the friction constant for the box in position a, G is the gravitational constant, and $3Gm_{box}$ is the downward force exerted by the stack of 3 boxes. The total acceleration can be calculated as

$$accel = \frac{F}{M} = \frac{F_{truck} - C_a \times 3Gm_{box}}{M}$$

$$accel = \frac{F_{truck}}{M} - C_a \frac{3Gm_{box}}{M}$$

Since M, G, and m_{box} are constants, let

$$k_0 = \frac{1}{M}$$

$$k_n = -\frac{nGm_{box}}{M} \text{ for } n \geq 1$$

We can then we can further simplify this equation to

$$a = k_0 F_{truck} + k_3 C_a$$

This is a linear function with independent variables F_{truck} and C_a. This function has a role corresponding to the term $k_0 F_{truck}$ that must be fulfilled by an object of type truck. It also has a role corresponding to the term $k_3 C_a$ that must be fulfilled by an object of type box.

In configuration 2, the frictional force is

$$F_{friction} = C_a \times 2Gm_{box} + C_b \times Gm_{box}$$

The total acceleration in configuration 2 is

$$accel = \frac{F_{truck} - C_a \times 2Gm_{box} - C_b \times Gm_{box}}{M}$$

$$accel = \frac{F_{truck}}{M} - C_a \frac{2Gm_{box}}{M} - C_b \frac{Gm_{box}}{M}$$

$$accel = k_0 F_{truck} + k_1 C_b + k_2 C_a$$

This is a linear function with independent variables F_{truck}, C_a, and C_b. This single function can cover both configurations 2 and 3 because the positions of the stacks are not important, only the distribution of blocks amongst stacks. There are 3 roles in this function, corresponding to the terms $k_0 F_{truck}$, $k_1 C_b$, and $k_2 C_a$.

#	Config.	Num. examples	Mode function
0	Noise	0	-
1	1	480	$v'_{truck} = v_{truck} + 0.025F_{push} - 7.5C_a$
2	2 and 3	960	$v'_{truck} = v_{truck} + 0.025F_{push} - 5.0C_a - 2.5C_b$
3	4	480	$v'_{truck} = v_{truck} + 0.025F_{push} - 2.5C_a - 2.5C_b - 2.5C_c$

Table 3.1: Modes learned for box pushing domain

Finally, the friction force in configuration 4 is

$$F_{friction} = C_a \times Gm_{box} + C_b \times Gm_{box} + C_c \times Gm_{box}$$

and the total acceleration is

$$accel = k_0 F_{truck} + k_1 C_a + k_1 C_b + k_1 C_c$$

This is a linear function with independent variables F_{truck}, C_a, C_b, and C_c. There are 4 roles in this function, one for each of the terms in the summation.

Therefore, we expect that our algorithm will learn 3 different modes for this domain, corresponding to the three equations derived above. The actual learned modes are shown in table 3.1. The first column in the table numbers the modes. The second column is the configuration the mode covers. This was determined by manually inspecting the training examples covered by the modes. The 0^{th} mode is the default noise mode that the system begins with. The third column lists the number of training examples assigned to each mode. In our training set there were 480 examples for each configuration, with 40 examples from each of the 6 permutations of box names in possible positions. The model didn't actually require so many examples to learn the correct modes, but we included all permutations to show that they are correctly unified under a single mode function each.

The fourth column shows the function learned for the mode. Here v'_{truck} is the predicted velocity of the truck in the next time step, and v_{truck} is its velocity in the current time step. The functions correspond to the ideal function we derived, and the proportions of the constants are as expected.

3.3 Relational state extraction

Our system combines positional information contained in the continuous state vector with object shape information into a scene graph, a 3D representation of the environment state. This is shown in the top right of figure 1.4. Positional information captured in the continuous state vector is updated every time step, but object shapes are only given once and assumed to be static. The system computes all spatial relationships between all objects in the scene graph using specialized algorithms. For example, it uses collision detection algorithms to determine if two objects intersect. Some example relations include $intersect(X, Y)$, $ontop(X, Y)$, and $east(X, Y)$. The set of truth values for all spatial relations between all objects makes up the relational state at each time step $\mathbf{r}_t$. The relational state is used for learning mode and role classifiers.

3.4 Mode Classification

The clustering component is responsible for identifying modes and mode functions. However, the mode functions alone are not sufficient to make predictions, because the agent doesn't know which mode to use at any time. The mode classification component is responsible for solving this problem. It learns a classification function $C(\mathbf{r}_t) \rightarrow m$ that predicts the correct mode m given the spatial relations $\mathbf{r}_t$ of the initial state, which we call the *relational state.* The relational state is computed from the continuous state vector and geometry information. Training examples for the classifier learner is a list

$$(\mathbf{r}_1, m_1), (\mathbf{r}_2, m_2), \ldots, (\mathbf{r}_t, m_t)$$

The mode m_t associated with each time step is obtained from the clustering step.

3.4.1 FOIL

First Order Inductive Learner (FOIL) (*Quinlan*, 1990) is the primary algorithm used to learn the classification function. We chose FOIL primarily for its simplicity of implementation; other inductive logic programming methods can be used by our system as well. FOIL learns first-order descriptions of object relationships. The input into FOIL are

1. A set of objects O
2. A set of relations R on those objects
3. A target relation T to be learned
4. A set of object tuples P serving as positive examples of T
5. A set of object tuples N serving as negative examples of T

FOIL outputs a set of Horn clauses, each describing a set of sufficient conditions for the target relation to hold. Each Horn clause has the form

$$T(X) \leftarrow L_1(X) \wedge L_2(X) \wedge \ldots \wedge L_k(X)$$

where T is the target relation and $L_1, L_2, \ldots, L_k$ are literals (either relations or negations of relations). X is a stand-in for any number of variables. An example Horn clause is

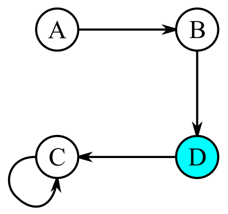
$$grandparent(X,Y) \leftarrow parent(X,Z) \wedge parent(Z,Y)$$

Algorithm 2 Sketch of FOIL

```
 1: procedure FOIL($relations, target$)
 2:     $pos \leftarrow PositiveTuples(target)$
 3:     $neg \leftarrow NegativeTuples(target)$
 4:     $disjuncts \leftarrow \emptyset$
 5:     while $|pos| > 0$ do
 6:         $clause \leftarrow \emptyset$
 7:         $neg' \leftarrow neg$
 8:         while $|neg'| > 0$ do
 9:             $lit \leftarrow ChooseLiteral(pos, neg')$
10:             $neg' \leftarrow FilterNegatives(neg', lit)$
11:             $clause \leftarrow clause \wedge lit$
12:         end while
13:         $pos \leftarrow FilterPositives(pos, clause)$
14:         $disjuncts \leftarrow disjuncts \vee clause$
15:     end while
16:     return $disjuncts$
17: end procedure
```

Algorithm 2 shows a pseudo-code version of FOIL. FOIL operates in two nested loops. In the outer loop (line 5), FOIL iteratively adds Horn clauses to the disjunction until all positive tuples of the target relation are covered. In the inner loop (line 8), FOIL iteratively adds literals to the right-hand-side of the Horn clause until all negative tuples of the target relation are excluded. The function $ChooseLiteral$ always chooses the literal that filters out the most negative tuples from neg'. The net result of the algorithm is that each clause covers a subset of the positive tuples of the target relation, and the entire disjunction together covers all positive tuples.

As another example, consider the diagram in figure 3.4. Suppose we wanted to learn a description of nodes that can reach the goal node D. The objects in this problem are the nodes A, B, C, and D. The relations are displayed in the table to the right. Note that only positive tuples for each relation are explicitly listed. By



relation	positive tuples
connected	(A,B), (B,D), (C,C), (D,C)
goal	(D)
can-reach	(A), (B), (D)

Figure 3.4: An example induction problem for FOIL.

convention, all tuples not explicitly listed as positive are negative. *can-reach* is the target relation. For this input, FOIL will learn the following clauses:

$$\textit{can-reach}(X) \leftarrow \textit{goal}(X)$$

$$\textit{can-reach}(X) \leftarrow \textit{connected}(X,Y) \wedge \textit{goal}(Y)$$

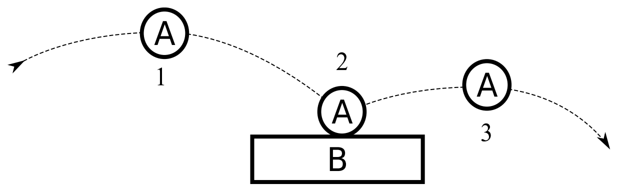
$$\textit{can-reach}(X) \leftarrow \textit{connected}(X,Y) \wedge \textit{connected}(Y,Z) \wedge \textit{goal}(Z)$$

Notice that the names of the nodes have been variablized in the learned description. This means that the description will apply to any set of nodes, regardless of names, as long as the tested relations hold between them.

3.4.2 Using FOIL to learn mode classifiers

To use FOIL to learn the mode classifier function $C(\mathbf{r}_t) \rightarrow m$, some tailoring in problem representation were required. First, FOIL learns binary classifiers, while the classification function must be able to distinguish between any number of modes. To address this problem, we use FOIL to learn binary classifiers for each pair of modes, so that we have $\frac{n(n-1)}{2}$ FOIL classifiers in total. We then use a one-against-one strategy (*Hsu and Lin*, 2002) to combine these binary classifiers into a single multi-class classifier. Basically, each binary classifier votes for one of two possible modes, and the mode with the most votes wins.

The second modification is the addition of a time parameter to each spatial re-



Time	Mode	Relations
1	1	`~intersect(A,B) & left(A,B)`
2	2	`intersect(A,B) & ~left(A,B)`
3	1	`~intersect(A,B) & ~left(A,B)`

Figure 3.5: An example of learning a binary mode classifier in FOIL.

lation. For example, $intersect(A, B)$ becomes $intersect(T, A, B)$. In this representation, $intersect(T, A, B)$ is true if and only if A intersects B at time step T. This modification is necessary because the training examples for our classifier learner spans multiple time steps, while FOIL has no concept of time. Adding the time parameter essentially collapses relations from different time steps into a single set.

The third modification is the addition of an artificial target relation. This relation is positive for all time steps in which one mode is exhibited and negative for the others. It also has a parameter for the object that we are learning the model for. The need for this second parameter is explained in more detail in section 3.6. The target relation has the form $target(T, X)$ where T is the time step and X is the target object. Basically, when learning a classifier to distinguish between modes m_1 and m_2, $target(T, X)$ relation marks all T during which X exhibited mode m_1.

Figure 3.5 shows an example of learning a binary mode classifier in FOIL. In the example, FOIL has three training examples, two from mode I, in which the ball is flying freely, and one from mode II, in which the ball is bouncing. After the representation transformation described above, the relations given to FOIL are:

relation	positive tuples
intersect	(2,A,B)
left	(1,A,B)
target	(1,A), (3,A)

In this case, the time steps where mode 1 hold are considered positive examples, and those where mode 2 hold are considered negative examples. Hence the positive tuples for the *target* relation are (1) and (3). The description FOIL learns for *target* is

$$target(T, X) \leftarrow \neg intersect(T, X, *)$$

This says that the ball exhibits behavioral mode 1 when it is not intersecting anything, otherwise it exhibits mode 2. The $*$ in the *intersect* literal indicates universal quantification.

3.4.3 Predicting modes

To predict the mode of a test example given the extracted relational state $\mathbf{r}_t$, the system evaluates each of the $\frac{n(n-1)}{2}$ binary classifiers on $\mathbf{r}_t$. For each binary classifier, each Horn clause in the classifier is evaluated on $\mathbf{r}_t$ until one matches or all of them have been tried. Because the clauses are first order, there can be many possible bindings of objects in the state to variables in the clause. For example, the first clause of the first binary classifier learned in the box pushing domain is $\textit{y-greater-than}(B, A) \wedge \textit{on-top}(C, B)$ (see table 3.2). A always binds to the target object, but B and C can potentially bind to any of the other boxes in the environment. Evaluating the clause on $\mathbf{r}_q$ therefore requires solving a constraint satisfaction problem (CSP), where each conjunct in the clause is a constraint. In our experiments, a basic backtracking solver using the Minimum Remaining Values heuristic (*Russell and Norvig*, 2003) was sufficient to solve all CSPs quickly. If a matching clause is found, then the binary classifier votes for the positive mode it was trained on. If none of the clauses match, then binary classifier votes for the negative mode. Each binary classifier votes for one of two modes, and the mode with the most votes is chosen.

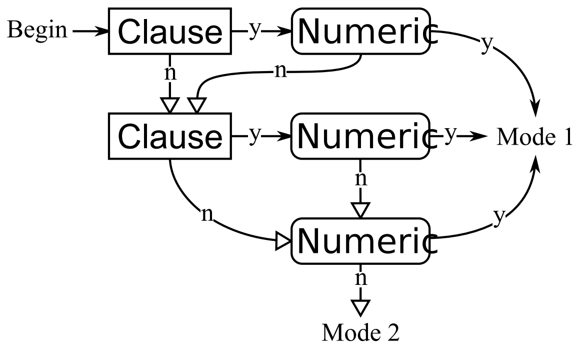


Figure 3.6: Classification process combining Horn clauses and numeric classifiers.

3.4.4 Combined symbolic and numeric classification

In some cases, the relational classifiers alone are not sufficient to distinguish two modes correctly. For example, a ball bouncing off a platform or rolling on it have the same relational state. The distinguishing feature is whether the ball's Y-velocity is zero or non-zero, which is not distinguished by any relations. When this is the case, a learned Horn clause may misclassify some negative examples as false positives. Furthermore, true positive examples that cannot be accurately described by Horn clauses will be classified as false negatives. To address this problem, our system learns a numeric classifier that distinguishes between the true and false positives of each clause whose false positive rate is above a hand-tuned threshold. Furthermore, if the false negative rate of the entire disjunction is too high, our system also learns a numeric classifier to distinguish between the true and false negatives. The final decision combines these two types of classifiers as shown in figure 3.6.

The algorithm has a waterfall model. If any pair of Horn clause/numeric classifiers both decide an instance is positive, then it is labeled as belonging to mode 1. Otherwise it goes on to the next clause/numeric classifier pair. If none of the pairs classify the instance as positive, then a final numeric classifier makes the decision between mode 1 and 2. When the false positive rate of a clause is low, the numeric classifier will not be learned and default to a "yes" answer. The same is true for false negatives.

Our system currently learns simple decision trees (*Mitchell*, 1997) for numeric classifiers. Each node in the tree compares the value of a particular dimension of the continuous state vector to a threshold and branches left if the value is below the threshold or right otherwise. Other methods such as support vector machines and linear discriminant analysis can also be used, but we chose decision trees for their simplicity and lack of tunable parameters.

We also found that more powerful learning methods are more prone to overfitting. Overfitting is especially prevalent in our system because it learns online, so training examples may be sampled from a narrow portion of the input space, while future test examples may come from an entirely different portion of the space. Furthermore, typical techniques for estimating out-of-sample error, such as cross-validation, are often ineffective in these contexts. Therefore, the system must rely on the intrinsic bias of the learning algorithm to avoid overfitting. Decision trees have a strong bias of separating data along axis-aligned hyperplanes that happened to suit our experimental domains, and therefore performed well in our experiments. But they will not work well in all domains. We also observed that the Horn clauses learned by FOIL are less vulnerable to overfitting than numeric classifiers. We hypothesize this is due to two reasons. First, the spatial relations the clauses are built from have strong biases that are well suited for spatial domains. Second, when FOIL overfits data, it tends to produce clauses that are conjunctions of large numbers of relationships, and situations that satisfy such conjunctions are sparse. Therefore, an overfit clause is unlikely to match future test examples and cause misclassifications. Instead, they will just be "dead weight."

As future work, we plan to investigate whether it is possible to learn new spatial predicates from numeric classifiers that prove to be accurate and useful over multiple domains.

3.4.5 Overcoming positive bias in FOIL

FOIL learns a binary classifier as a disjunction of clauses. Each clause covers a portion of the positive training examples, and new clauses are added until all positive training examples are covered. The space of negative examples is then considered to be whatever is not covered by the clauses. This strategy has an intrinsic bias that favors problems where the space of positive examples is simpler to describe than the space of negative examples. In some situations, the simplicity of the descriptions can be extremely asymmetric, even to the point where it is impossible to describe one set of examples generally using clauses of spatial relations while the other set of examples can be characterized using a single relation.

In learning a binary classifier in our system, one mode is considered the positive class while the other is considered the negative class. If the wrong mode is chosen for the positive class, the resulting classifier will likely overfit the data and not generalize very well. To accommodate these problematic cases, our system learns two binary classifiers for each pair of modes (i, j), one in which mode i is used as the positive class and one in which mode j is used as the positive class. The system then keeps the classifier with the lower training error.

3.4.6 Mode classifiers learned in the box pushing domain

Table 3.2 shows the mode classifiers learned by our system. The table is split into three sections, one for each binary classifier learned for each pair of modes. For example, the first section describes the binary classifier learned to distinguish between modes 1 and 2. A positive output from this classifier is a vote for mode 1, whereas a negative output is a vote for mode 2. The mode numbers correspond to the numbers in the left-most column of table 3.1.

Within each section is the list of clauses comprising each binary classifier, along with the number of training examples they correctly classify and those that they misclassify. In the mode 1 versus mode 2 case, a correctly classified training example is one that was assigned to mode 1 by the clustering algorithm and which satisfies the clause. An incorrectly classified training example is one that was assigned to mode 2 by clustering, but still satisfies the clause. The "train perf." value is the proportion of training examples the classifier as a whole, including any numeric classifiers, correctly labels.

The last column in the table indicates whether a numeric classifier was learned for the clause. In this domain, the spatial relations alone were sufficient to perfectly distinguish all modes, so no numeric classifiers were learned.

The last row in each section, labeled "Negative," represents the "fall-through" examples that are not covered by any of the clauses. Assuming FOIL learned the clauses correctly, these examples should be labeled with the negative mode (mode 2 in this case). An incorrectly classified example in this row is one that is assigned to mode 1 by clustering and not covered by any of the clauses.

Within each clause, the individual literals representing spatial relation tests are variablized. The variable A is always bound to the target object (the truck in this case), while all other variables can bind to any object. For clarity, we have omitted the time step parameter that is normally tested by each spatial relation (see section 3.4.2). Variables other than A serve as existential quantifiers. For example, in the first clause, *y-greater-than*(B, A) is satisfied as long as there exists some object that is higher than A. However, since the clauses are first order, B must be bound consistenly throughout. So the first clause is only satisfied if there is an object B above A that also has another object C on top of it.

A manual inspection of the clauses suggests that it aligns with our intuition of how to distinguish the modes from each other. The classifier learns that mode 1 is distinguished from mode 2 by the presence of a stack of 3 boxes (C is on top of B and B is higher than A). Both modes 1 and 2 are distinguished from mode 3 by the

Clause	**Num. correct**	**Num. incorrect**	**Numeric classifier?**
Mode 1 vs. 2 (train perf. 1.0000)			
$y\textit{-greater-than}(B, A) \wedge \textit{on-top}(C, B)$	480	0	No
Negative	960	0	No
Mode 1 vs. 3 (train perf. 1.0000)			
$y\textit{-greater-than}(B, A) \wedge intersect(B, C)$	480	0	No
Negative	480	0	No
Mode 2 vs. 3 (train perf. 1.0000)			
$y\textit{-greater-than}(B, A) \wedge intersect(B, C)$	960	0	No
Negative	480	0	No

Table 3.2: Learned mode classifier for the box pushing domain.

presence of an object positioned higher than the truck.

3.5 Role classification

When applying mode functions to test states with different objects than the training data, the system must assign the correct object to each role, i.e. solve the role assignment problem. For each role, our system learns a relational classifier that can determine if an object is suitable for the role, which we call a role classifier. Like mode classifiers, role classifiers are disjunctions of first order Horn clauses learned using FOIL.

This process is illustrated in figure 3.7. The top-right portion of the figure is a reproduction of figure 3.3. As shown in the figure, the training data given to FOIL consist of the relational states of each training example, and the object-to-role mappings obtained from linear regression. These two pieces of data are combined so that FOIL knows which object fulfilled each role in each training example, and the spatial relationships between that object and all other objects. From this FOIL learns a set of first order Horn clauses that distinguishes the role-fulfilling object from other objects in the environment.

More formally, for each role ρ, the input into FOIL is

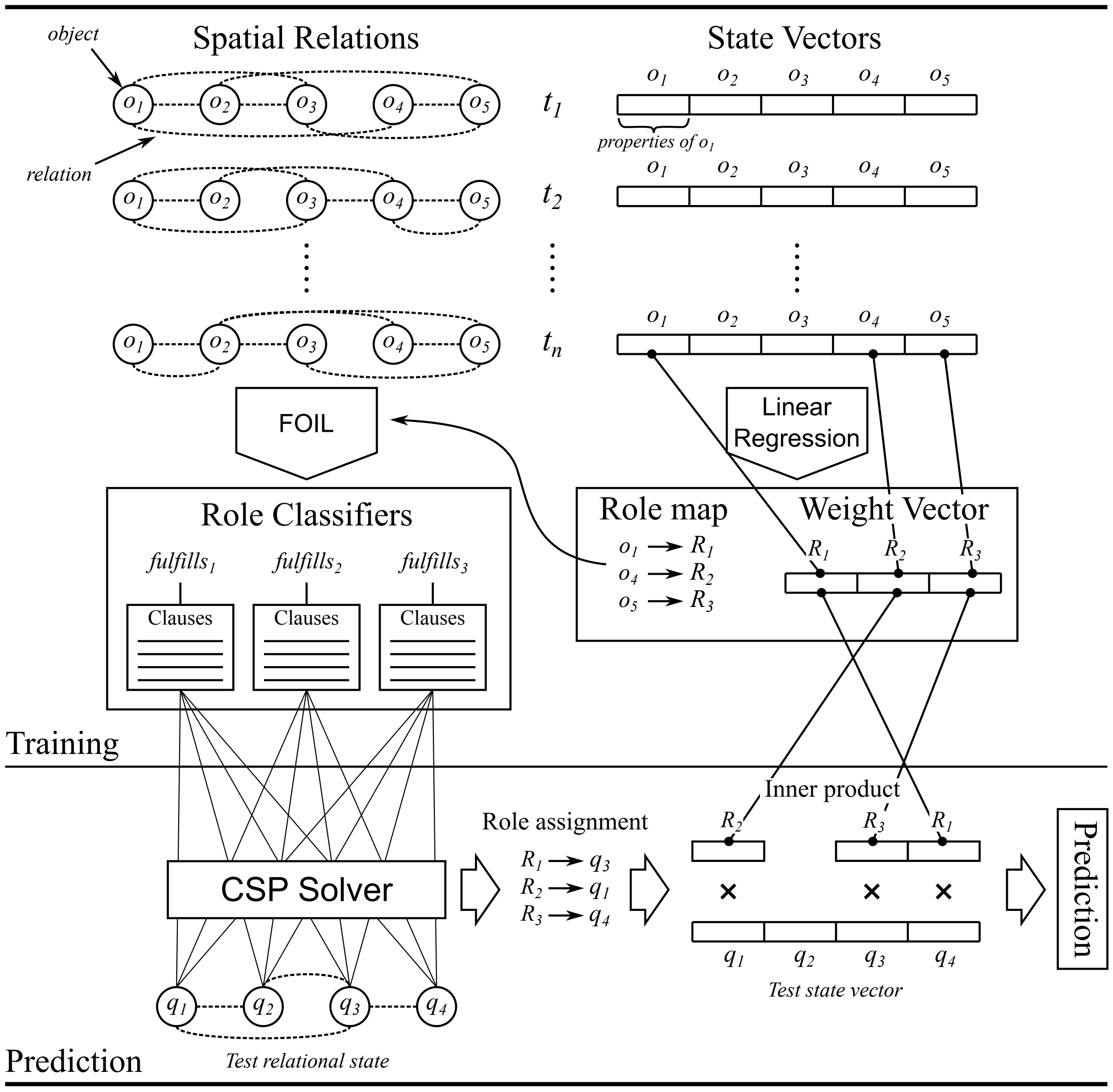


Figure 3.7: Learning role classifiers to solve the role assignment problem.

- All spatial relations in the relational states $(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n)$, augmented with the time value as described in section 3.4.1.

- A special relation with the form $fulfills_\rho(T, X, Y)$, where T is the time step, X is the target object, and Y is any other object. $fulfills_\rho(T, X, Y)$ is true if and only if X is the target object and Y fulfills role ρ at time T.

Note that unlike the mode classifier which uses a one-against-one strategy, the role classifier uses a one-against-all strategy. This means that all objects that do not fulfill the role being learned are negative examples, and only the object that fulfills the role is a positive example.

The output from FOIL is a set of clauses that establish the requirements for an object to fulfill a role. For instance, the classifier for one of the balls in the two ball bouncing example (section 1.2.5) may be:

$$fulfills_\rho(T, X, Y) \leftarrow \textit{is-ball}(T, Y) \wedge intersect(T, X, Y)$$

In other words, the object that fulfills the role must be of type *ball* and intersect the target ball. The other ball role in the mode is the role for the target object, so it is automatically identified and bound to X.

When the mode function is applied to a new state during prediction, the system checks if each object satisfies the $fulfills_\rho$ relation for each role. This requires searching for a valid assignment of objects to the variables in the first-order description, which is accomplished using the same CSP solver as for mode classification. If a complete assignment is found, then the role is assigned to the object. This process is illustrated at the bottom of figure 3.7. In the previous example, there are no unbound variables, so no search is required: each object is simply checked to see if it

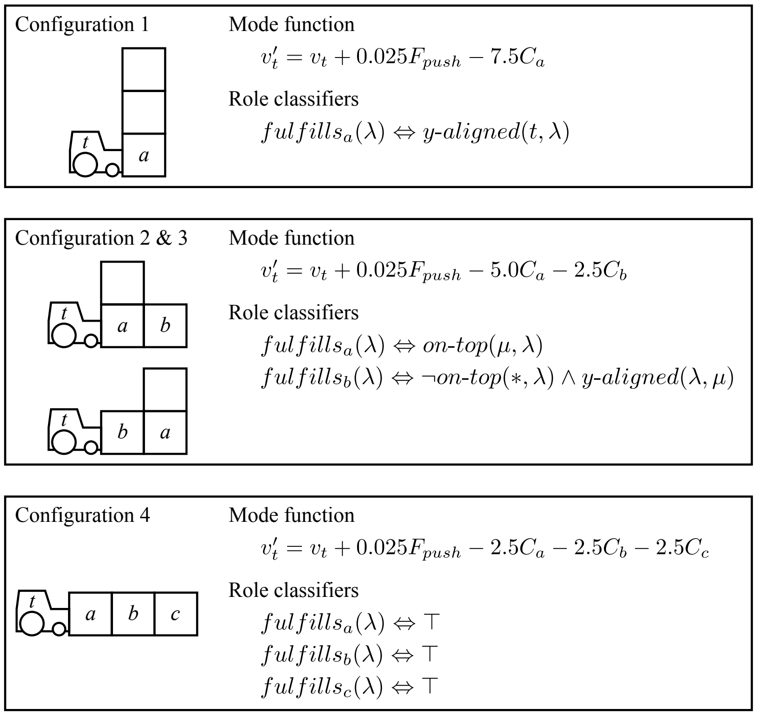

Configuration 1
Mode function
$v'_t = v_t + 0.025F_{push} - 7.5C_a$
Role classifiers
$fulfills_a(\lambda) \Leftrightarrow \textit{y-aligned}(t, \lambda)$
t
a
Configuration 2 & 3
Mode function
$v'_t = v_t + 0.025F_{push} - 5.0C_a - 2.5C_b$
Role classifiers
$fulfills_a(\lambda) \Leftrightarrow \textit{on-top}(\mu, \lambda)$
$fulfills_b(\lambda) \Leftrightarrow \neg\textit{on-top}(*, \lambda) \wedge \textit{y-aligned}(\lambda, \mu)$
t
a
b
t
b
a
Configuration 4
Mode function
$v'_t = v_t + 0.025F_{push} - 2.5C_a - 2.5C_b - 2.5C_c$
Role classifiers
$fulfills_a(\lambda) \Leftrightarrow \top$
$fulfills_b(\lambda) \Leftrightarrow \top$
$fulfills_c(\lambda) \Leftrightarrow \top$
t
a
b
c

Figure 3.8: Role classifiers learned for the box pushing domain.

is a stair and intersects the ball. This is the case in most of our experiments, so the process is fast. However, in the general case, a first-order description may contain free variables, and backtracking search would be necessary. Finally, if all roles are successfully assigned to objects, the mode function is applied on the vector consisting of only their properties. If some roles are not assigned, the system assumes that the mode is incorrect and will try another mode.

3.5.1 Role classifiers learned in the box pushing domain

Figure 3.8 shows the role classifiers our system learned for each mode of the box pushing domain. The role classifiers for the truck roles were omitted because they were trivial, i.e. there is only one truck in the domain to fulfill the role. For configuration 1, the system learns that the box that should fulfill role a is the one that is vertically (y) aligned with the truck. This makes sense as the other two boxes are above the truck. For role a in configuration 2, the role classifier requires that the fulfilling box have something be on top of it. For role b, the role classifier requires that the fulfilling box have nothing on top of it and that it's vertically aligned with the truck. The second conjunct rules out the box that is above a. For configuration 3, because any box can correctly fulfill any role, the role classifiers are empty.

3.6 Prediction using the learned model

Finally, we describe how a prediction is made by a learned model. The prediction algorithm is shown in figure 1.5. Given a query state vector $\mathbf{x}_q$, our system first extracts the relational state $\mathbf{r}_q$ in the same way as it did during training. The system next uses the mode classifier to determine the mode to use for prediction, as discussed in section 3.4.3. After the appropriate mode is chosen, the system assigns the roles of the mode function to the correct objects in the environment. This establishes which object properties to use when evaluating the mode function. Finally, the mode function is evaluated with the appropriate object properties to produce the final prediction.

3.7 Discussion

The models learned by our system combine the benefits of propositional and relational representations. Generally speaking, our algorithm takes simple linear models learned using standard linear regression techniques and generalizes them with relational representations in two ways:

1. The propositional representation of the linear function is generalized into a first-order representation with roles. Instead of corresponding to specific dimensions of a propositional state, the weights in our mode functions correspond to properties of roles, and roles can be bound to different objects depending on context. So a linear function describing the velocity of a ball exiting a bounce can be

used for any object the ball bounces off of.

2. The total behavior of the environment is modeled as a combination of linear mode functions, conditioned on the relational mode classifier. When the ball is flying through the air, one linear function is used to describe its behavior. When it is bouncing against the ground, another function is used to describe its behavior. While many existing methods use this approach, as discussed in the related works chapter, none of them switch between modes based on the relationships between objects.

CHAPTER IV

Experiments and Results

In the previous chapter, we described the box pushing domain and the models our system learned in that domain to aid in exposition. This chapter describes experiments we performed to evaluate our model learning algorithm and their results. We begin with a description of the environment used in the experiments, and then go on to discuss results obtained in increasingly more complex variations of that environment.

4.1 The ball-box-ramp domain

This environment consists of a ball, box, and ramp inside a room with 2 walls, a floor, and a ceiling. We use the Chipmunk physics engine (*Lembcke*, 2013) to simulate realistic interactions between the ball and other objects. The positions of the ramp, box, floor, walls, and ceiling are fixed, but the ball is free to move about. The ball undergoes three major types of motions: flying, colliding, and sliding. When the ball is flying, it is not interacting with any other objects, and the only force on it is gravity. When the ball collides with an object, its velocity is reflected perpendicular to the surface of the collision and damped according to the elasticity constant of the object. When the ball slides against a surface, its velocity is damped according to the Coulomb model of friction. The Coulomb friction model applies a constant deceleration in the opposite direction of the ball's motion. The magnitude of the deceleration is affected by the friction constant of the surface, the incline of the surface, and the mass of the ball. The simulator is not able to simulate ball rolling behavior, so we did not consider rolling in our experiments. Note that the agent does not execute actions that affect the environment in any way. Its only task is to observe environment dynamics and learn a model.

The continuous state of the environment consists of the position, rotation, and scaling of each object, as required from all environments. In addition, the ball's x and y velocities are included in the state vector. The velocities are calculated by taking the difference in the x (y) coordinates of the ball in the current time step from those in the previous timestep. The system used the typical relations when computing the relational state.

The purpose of these experiments is to show that our algorithm can learn accurate models of realistic physical interactions between rigid objects. Physics simulations provide a variety of different modes that arise from interactions of objects with different shapes and properties such as mass, elasticity, and friction constants. This allows us to test the performance of the segmentation system. Furthermore, the conditions under which different interactions occur are non-trivial to learn, usually requiring conjunctions of several spatial relations. This allows us to test the performance of the classification system.

4.2 Experimental procedure

The model is trained on a sequence of blocks. In each block, the environment is initialized in a particular configuration, and then the physics simulation runs its course for 200 time steps and thus produces 200 training examples. The initial configurations of the scenarios are generated by varying a number of parameters so that the agent is exposed to many qualitatively distinct interactions. If we instead only exposed the

Parameter	Possible values
Ramp placement	Left of box, right of box
Ramp and box touching	Yes, no
Ball position	Leftmost, above left object, between objects, above right object, rightmost
Ball's initial direction of travel	Left, right

Table 4.1: Environment configuration parameters

agent to a single initial configuration that was run for more time steps, the simulation would converge to a resting state where no interesting transitions are generated. The configuration parameters are shown in table 4.2, and some example configurations are shown in figure 4.1. There are 40 distinct configurations in total.

In addition to the configuration parameters, other factors that affect the environment are:

1. The random seed. While the configuration parameters establish the relative positions of objects in the environment, their exact locations and sizes are randomly varied. The random seed controls this variation.

2. Friction and elasticity constants.

3. Ramp incline.

Each distinct value of friction and elasticity constants results in different interactions between the ball and other objects, which in turn results in different modes. Recall that we assume that objects of the same type behave identically. To satisfy this assumption, each distinct combination of object shape, friction constant, and elasticity constant must be a distinct type. For example, the box will have type `box_0.1_0.9` if its friction constant is 0.1 and its elasticity constant is 0.9. The ramp will have type `ramp_0.3_0.8_60` if its friction constant is 0.3, elasticity constant is 0.8, and incline is 60 degrees. We encode the parameter values into the type names because it guarantees unique names and aids debugging; the learning algorithm does

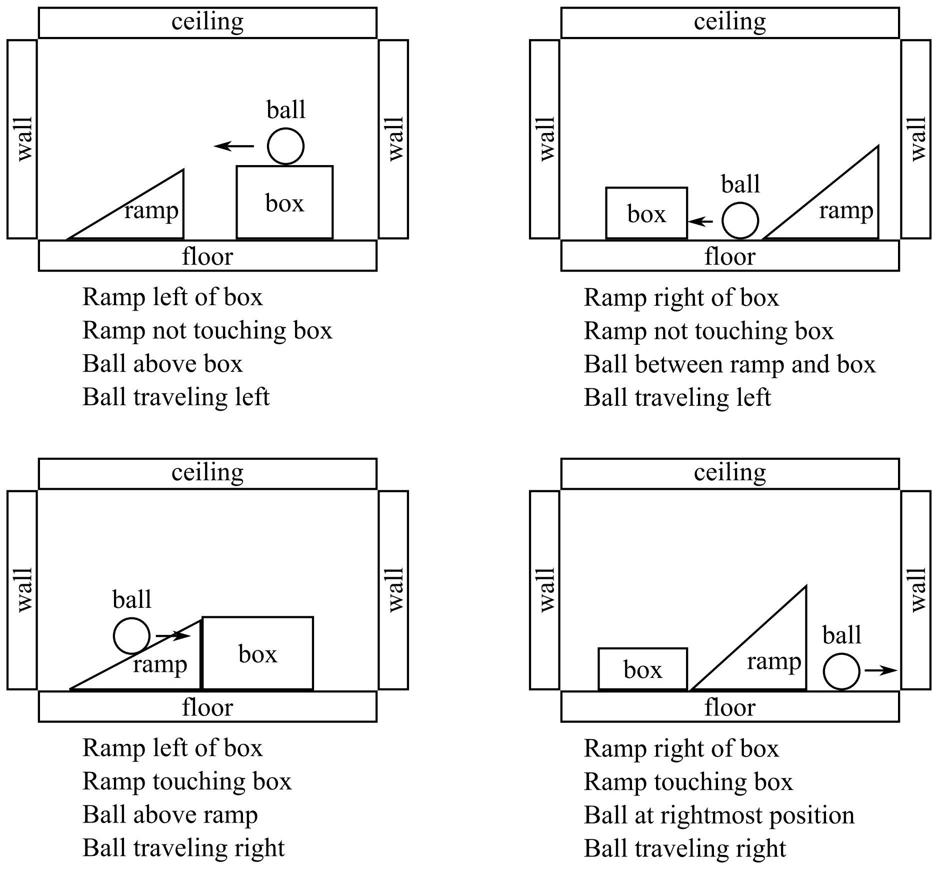


Figure 4.1: Some possible initial configurations of the physics simulation domain

not extract parameter values from the names or interpret them in any other way. Therefore, the unique types prevents the learned model from incorrectly generalizing the modes covering the ball's interactions with objects having one friction constant from objects having another friction constant, but the actual functions associated with those modes still have to be learned without prior knowledge.

A major consequence of using online learning algorithms is that the presentation order of the configurations in the training sequence affects the accuracy and generality of the final models. For example, suppose that all objects in the environment have no friction, so that a ball sliding on a horizontal surface maintains a constant x velocity. In this case, the most general mode the system can learn is $v'_x = v_x$, where v_x is the ball's x velocity in the current time step, and v'_x is its x velocity in the next time step. Now consider two different orderings for training scenarios. If the learner is first presented a scenario with a large number of examples of the ball moving with constant velocity k_1, it will learn a mode with the form $v'_x = k_1$. Later, the learner may get a scenario with a large number of examples of the ball sliding at a different constant velocity k_2, and learn a separate mode $v_x = k_2$. These modes can only correctly predict the ball's x velocity if it were sliding at velocity k_1 or k_2, but nowhere else. On the other hand, if the learner first experiences a scenario where the ball bounces back and forth, it will have examples of the ball moving at different velocities and will learn the mode $v_x = v'_x$, which correctly predicts x velocity for all constant velocities. Fortunately, this example will not actually occur in practice because the mode unification algorithm will correctly merge the two overspecific modes into the general mode $v_x = v'_x$. However, there are more complex cases where mode unification fails and the system cannot recover from a lack of diversity in some window in the training regime.

To address this concern, we average our results across many distinct random configuration orders. The order of the examples within a single block is not randomized, as this better mimics online data gathering, where the agent cannot uniformly sample the entire space of examples but must move through the space by taking local steps. In the following experiments, our results are averaged over 10 different block orderings, and each ordering is repeated 5 times with different random seeds, for a total of 50 training batches.

In the following sections, we describe 2 experiments on this domain. In the first experiment, surfaces will not have any friction, and all training and test examples will share the same elasticity constants and ramp inclines. In the second experiment, surfaces will have friction, but all training and test examples will still share the same friction constants, elasticity constants, and ramp inclines. In the third experiment, we subject the learning algorithm to objects with different friction constants. This will result in the model taking on significantly more modes, and serves as a demonstration that our algorithm is capable of simultaneously learning models over different conditions.

4.3 Experiment 1

We begin with the simplest version of the ball-box-ramp environment in which all objects are frictionless and have the same elasticity constant of 0.9, and the ramp is always inclined at $30°$. With this setup, the model has to learn a minimum number of modes while all the learning mechanisms are still exercised. This allows us to analyze in detail how the algorithm behaves in a real learning scenario.

As discussed previously, each training batch consists of a number of training blocks, each starting with a different initial configuration and running for 200 time steps. Since there are 40 different initial configurations, the model should be trained on $40 \times 200 = 8000$ training examples in a single batch. However, we found that examples from certain modes occur relatively rarely in simulation, such as examples of the ball bouncing against a wall. Training on the 40 blocks once doesn't present the model with a sufficient number of training examples from these rare modes, so we repeat each block twice with different random seeds. Therefore, the model is trained on 80 blocks and 16000 examples in total.

4.3.1 Learned modes

We first analyze in detail the modes learned by the model to better understand how the learning algorithm works on real data. Because this is not a quantitative analysis, averaging over the training batches does not make sense. Therefore, the learned modes shown here are from a single training batch, chosen as the most "correct" and representative of all the batches. Some of the modes learned in most other batches had trivial variations from the ones shown here, such as learning the modes in a different order. A few had important variations such as missing or over-specific modes. The variations are a consequence of using online algorithms and varying the order of training blocks.

The learned modes are shown in table 4.2. The first column in the table numbers the modes. The second column of the table labels each mode with the behavior we think it is trying to describe. This was determined by manually inspecting the training examples covered by the modes. The third column lists the number of training examples assigned to each mode. The fourth column shows the function learned for the mode. In the functions, v_x and v_y are independent variables corresponding to the x and y velocities of the ball in the current time step, and v'_x and v'_y are the predicted velocities in the next time step. A manual inspection of the functions confirm that they make sense. For example, when the ball is flying through the air, its x velocity does not change (there is not air resistance), while its y velocity decreases at a constant rate based on the gravitational constant. This is consistent with the functions $v'_x = v_x$ and $v'_y = v_y - 9.8 \times 10^{-4}$. It also makes sense that the bouncing modes are $v'_x = -0.9v_x$ and $v'_y = -0.9v_y$, since we set the elasticity constants of all objects at

#	**Behavior**	**Num. examples**	**Mode function**
x velocity model			
0	Noise	7	-
1	Slide on horiz. surf./fly	15574	$v'_x = v_x$
2	Bounce on vert. surf.	295	$v'_x = -0.9v_x$
3	Bounce/slide on ramp	124	$v'_x = 0.525v_x - 0.823v_y + 4.24 \times 10^{-4}$
y velocity model			
0	Noise	10	-
1	Fly (in gravity)	9169	$v'_y = v_y - 9.8 \times 10^{-4}$
2	Slide on horiz. surf.	6355	$v'_y = 0$
3	Bounce on horiz. surf.	334	$v'_y = -0.9v_y$
4	Bounce/slide on ramp	132	$v'_y = -0.823v_x - 0.425v_y - 2.45 \times 10^{-4}$

Table 4.2: Modes learned for x and y velocities of ball in environments with no friction, 0.9 elasticity, and 30° ramps.

0.9, and the negation indicates a reflection in the direction of travel after the bounce. Furthermore, note that all the functions only depend on v_x and v_y, meaning they are independent of the positions of any objects and even of the ball itself. All these factors suggest that the segmentation algorithm has produced the most general set of modes possible for this environment.

4.3.2 Learned mode classifier

We now discuss in detail the classifiers learned to distinguish the x and y velocity modes. As described in section 3.4, the mode classifier predicts the mode the modeled quantity will exhibit in the next time step given the relational and continuous states of the current time step. The mode classifier is a multi-class classifier composed of $N(N-1)$ binary classifiers, each trained to discriminate between two modes. Each binary classifier is learned with FOIL and composed of a disjunction of clauses, with each clause being a conjunction of variablized spatial relations. A clause whose false positive rate is too high suggests that the preconditions for a mode are not sufficiently captured by spatial relations, so a decision tree classifier that tests the continuous state is learned to augment the clause.

Clause	**Num. correct**	**Num. incorrect**	**Numeric classifier?**
Mode 1 vs. 2 (train perf. .9984)			
$\neg intersect(A,*)$	9016	0	No
$\neg y\text{–}aligned(A,*)$	6464	0	No
$x\text{–}aligned(A,B) \wedge ramp(B)$	57	0	No
$\neg y\text{–}greater\text{–}than(A,*) \wedge floor(B)$	11	0	No
Negative	295	26	No
Mode 1 vs. 3 (train perf. .9964)			
$\neg intersect(A,*)$	9016	0	No
$\neg y\text{–}aligned(A,*)$	6464	0	No
$x\text{–}greater\text{–}than(B,A)$	16	0	No
$x\text{–}greater\text{–}than(A,B)$	8	0	No
$\neg ramp(*)$	13	0	No
Negative	124	57	No
Mode 3 vs. 2 (Negated) (train perf. 1)			
$x\text{–}aligned(A,B) \wedge ramp(B)$	124	0	No
Negative	295	0	No

Table 4.3: Clauses learned for x velocity classifier

Table 4.3 shows the binary classifiers learned for the model of x velocity after the learning algorithm has seen all 16000 training examples. The table follows the same style as table 3.2 explained in section 3.4.6. The only new entity is the special variable $*$ which serves as an universal quantifier. For example, $\neg intersect(A,*)$ is only satisfied if $\forall_B \neg intersect(A,B)$, or in other words, the target object doesn't intersect any other object.

Interpreting the learned clauses is instructive for what FOIL is actually doing. Consider the binary classifier that distinguishes between modes 1 and 2. The first clause, $\neg intersect(A,*)$, states that if the ball is not intersecting anything, then it is in mode 1 because it is flying. The second clause, $\neg y\text{–}aligned(A,*)$, covers cases where the closest object to the ball does not occupy the same vertical space. For example, if the ball is bouncing against a wall, then the wall would be $y - aligned$ with it and this would not be a case of bouncing against a vertical surface rather than horizontal sliding. If the ball is bouncing against a ramp, the ramp would also be *y–aligned*

Clause	**Num. correct**	**Num. incorrect**	**Numeric classifier?**
Mode 1 vs. 2 (train perf. .9281)			
$x\text{–}greater\text{–}than(B, A)$	145	0	No
$x\text{–}greater\text{–}than(A, B)$	129	0	No
Negative	124	21	No

Table 4.4: Binary classifier learned using mode 2 as the positive class.

with it. On the other hand, if the ball is sliding on a horizontal surface such as the top of the box or floor, then that object's greatest y coordinate will be slightly below the ball, and it will not be *y–aligned*. The third clause, $x\text{–}aligned(A, B) \wedge ramp(B)$, catches edge cases where the ball bounces against the ramp at a certain angle such that its x velocity doesn't change as it should in the majority of collisions with the ramp. Notice that because this binary classifier is only responsible for distinguishing between modes 1 and 2, it doesn't have to add extra literals to this clause to rule out mode 3 (ramp bouncing) examples, which would otherwise be included under this clause. The last clause, $\neg y\text{–}greater\text{–}than(A, *) \wedge floor(B)$, catches another edge case where ball has penetrated into the floor during a bounce due to imperfections in the physics simulation. If the physics simulator were perfect, the ball would never penetrate the ground, and these examples would fall under the second clause. As it stands, the penetration is deep enough that the ball is considered *y–aligned* with the ground, so an extra clause is needed to cover the examples.

The last binary classifier, mode 3 versus 2, is a negated classifier. This is because the set of clauses required to cover the mode 3 examples is simpler and more accurate than that needed to cover the mode 2 examples: only a single clause is needed to perfectly distinguish between the two modes. On the other hand, table 4.4 shows the non-negated binary classifier. It requires two clauses and has a higher training error. This is an example of asymmetry in simplicity of description that was discussed in section 3.4.5.

Table 4.5 shows the classifier learned for the y velocity model. The most notable

Clause	**Num. correct**	**Num. incorrect**	**Numeric classifier?**
Mode 1 vs. 2 (train perf. 0.9978)			
$\neg intersect(A,*) \wedge y\text{–}aligned(A,B)$	6702	0	No
$\neg x\text{–}aligned(A,*)$	442	0	No
$\neg intersect(A,*) \wedge \neg floor(*)$	473	0	No
$\neg intersect(A,*)$	1428	16	No
$\neg y\text{–}greater\text{–}than(A,*)$	69	0	No
Negative	6537	19	No
Mode 1 vs. 3 (train perf. 0.9971)			
$\neg intersect(A,*)$	8957	0	No
$\neg x\text{–}aligned(A,*)$	88	0	No
$\neg y\text{–}aligned(A,*)$	16	0	No
$\neg ramp(*)$	13	0	No
Negative	124	72	Yes
Mode 1 vs. 4 (train perf. 0.9981)			
$\neg intersect(A,*)$	8957	0	No
$\neg x\text{–}aligned(A,*)$	88	0	No
$ramp(B)$	59	0	No
$wall(B)$	6	0	No
Negative	183	18	No
Mode 3 vs. 2 (Negated) (train perf. 1.0000)			
$\neg y\text{–}greater\text{–}than(A,*)$	113	0	No
$y\text{–}aligned(A,B) \wedge x\text{–}aligned(A,B)$	11	0	No
Negative	6553	0	No
Mode 2 vs. 4 (train perf. 0.9978)			
$y\text{–}greater\text{–}than(A,B) \wedge$ $y\text{–}aligned(A,C)$	211	3	No
$\neg y\text{–}aligned(A,*) \wedge \neg on\text{–}top(A,*)$	228	0	No
$\neg y\text{–}aligned(A,*) \wedge floor(B)$	6114	12	No
Negative	168	0	No
Mode 3 vs. 4 (Negated) (train perf. 0.9967)			
$ramp(B)$	124	1	No
Negative	182	0	No

Table 4.5: Clauses learned for y velocity classifier

Clause	**Num. correct**	**Num. incorrect**	**Numeric classifier?**
Mode 2 vs. 3 (train perf. 0.9988)			
$\neg y\text{–}aligned(A, *)$	6342	0	No
$x\text{–}greater\text{–}than(B, A)$	104	0	No
$x\text{–}greater\text{–}than(A, B)$	99	0	No
Negative	124	8	No

Table 4.6: Non-negated binary classifier for y velocity modes 2 and 3.

feature is that the binary classifier for modes 2 and 3 is negated. Here the "positive" mode is 3, the mode of the ball bouncing against the ground, and the "negative" mode is 2, the mode of the ball sliding on a horizontal surface. It was easier for FOIL to identify the bouncing mode (3) by testing for the penetration of the ball into the surface it bounces off of with $\neg y\text{–}greater\text{–}than(A, *)$, which covers a majority of the examples. The 11 examples covered by the second clause, $y\text{–}aligned(A, B) \wedge x\text{–}aligned(A, B)$, are not of the ball bouncing, but rather the edge case where the ball slides from a flat surface to the lip of the ramp. Since the ball's y velocity increases as a result of hitting the ramp, the system considers it a bounce. This is arguably incorrect, but these cases occur so rarely (only 11 examples in a total of 16000) that they don't warrent the model to learn a new mode. The rest of the examples are correctly classified by the fall-through condition as the rolling mode (mode 2).

Table 4.6 shows the non-negated binary classifier for modes 2 and 3. It has one more clause than the negated version, making it more complex. It also incorrectly classifies 8 examples, whereas the negated version does not misclassify any examples. Furthermore, it's not apparent why the dual clauses $x\text{–}greater\text{–}than(B, A)$ and $x\text{–}greater\text{–}than(A, B)$ are learned, and whether they would generalize correctly to test examples or are simply overfitting the training data.

Another notable feature in table 4.5 is the numeric classifier learned for the negative case of modes 1 and 3. Recall that the numeric classifier is learned when a clause's false positive rate is too high, or the binary classifier's false negative rate is

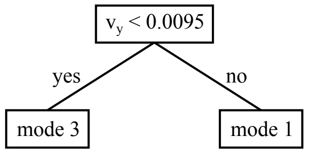


Figure 4.2: Decision tree learned to distinguish between modes 1 and 3 in the y velocity model.

too high, which is this case. Figure 4.2 shows the decision tree learned as the numeric classifier. However, further inspection of why the symbolic clauses do not cover all positive cases correctly shows that all the errors are due to imperfections in the physics simulator. Therefore, the decision tree doesn't make sense. More meaningful numeric classifiers will be discussed in experiment 2.

4.3.3 Prediction accuracy

We now present results on the prediction accuracy of the learned models. 24000 testing scenarios are generated in the same way as the training scenarios, but using three different random seeds from the training examples. This means that even though the training and testing examples are qualitatively similar, the absolute positions and sizes of objects in each are different. Furthermore, the entire configuration in each training and testing scenario is shifted in a random direction by a random distance, so that the absolute positions of objects can vary significantly.

At specific intervals in this training sequence, we take the model learned up to that point and test its accuracy on all 24000 test examples. The model does not learn during testing. The intervals we chose are after the 1st, 16th, 32nd, 48th, 64th, and 80th training blocks. As previously mentioned, the results are averaged over 10 different training block orders and 5 different sets of random seeds, for a total of 50 training batches.

Figure 4.3 shows the overall accuracy of the models learned for predicting x and y velocities. The x axis marks the number of training scenarios given to the model.

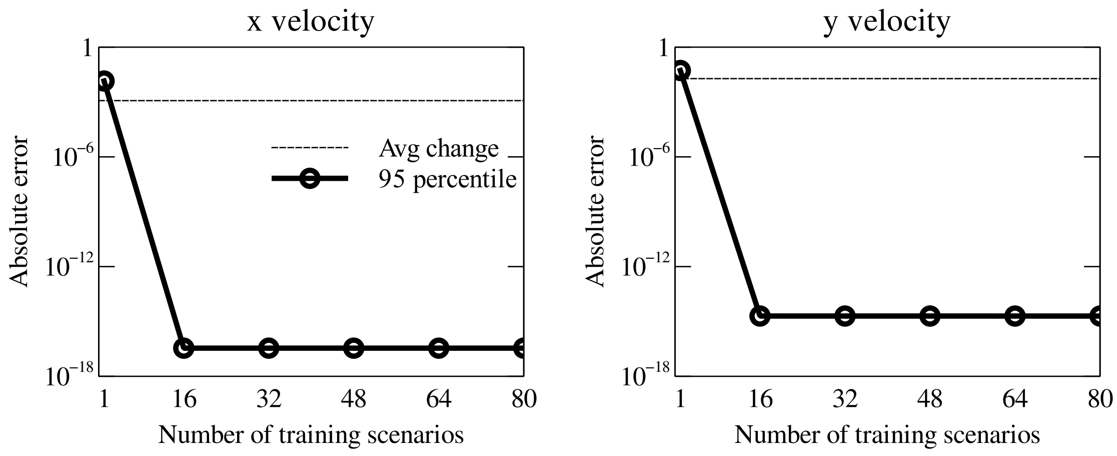


Figure 4.3: Prediction accuracy for x velocity and y velocity models

The numbers on the x axis can be multiplied by 200 to obtain the number of training examples given to the model. The y axis marks the absolute error of the prediction and is in log scale. The thick line indicates the 95^{th} percentile of the prediction errors made by the model, meaning 95% of the errors fall below the line. The dashed line that runs across the entire graph represents the average magnitude of change in the predicted property. This is to give a sense of scale for how much the property is varies, and what would be considered an accurate prediction.

Figure 4.3 suggests that the learned models are very accurate, such that after seeing just sixteen training scenarios, they achieve a prediction error of less than 1.0×10^{-12} on at least 95% of the test examples. Informal experiments suggest that comparable error rates are actually achieved much sooner, after just two or three training scenarios. These plots are misleading: the test examples are unevenly distributed across the different modes, with the majority of the examples coming from the simplest modes. For the x velocity model, the flying/horizontal sliding mode where the x velocity doesn't change takes up 23495 of the 24000 examples. For the y velocity model, the flying mode where the y velocity is constantly reduced by gravity and the horizontal sliding mode where the y velocity is always 0 make up 23660 of the 24000 examples together. Since these modes are easy to learn, the models can predict them with near zero error after training on just a few scenarios, so the majority of the prediction errors are close to 0. These easy predictions obscure how the models perform on more difficult modes.

To visualize how the model performs on other modes, we graph the prediction errors separately for each mode. This is shown in figures 4.4 and 4.5. We switch to box-and-whisker plots for these figures to better depict the distribution of errors. The boxes in each plot span the first and third quartiles of the errors, and the whiskers span the 5th and 95th percentiles. The horizontal line inside each box indicates the median. The dashed horizontal lines represent the average magnitude of change of the velocity within each mode. Note that the models' prediction errors on the horizontal/vertical bouncing modes and the modes involving the ramp remain high for many more training scenarios. One reason for this is that there are simply too few training examples for these modes in the entire training set, so the model doesn't see enough examples to learn the modes until it has trained on many more scenarios. The other reason is that the classifiers for these modes are difficult to learn, and generalize poorly to qualitatively different scenarios that the models have not been trained on. In any case, the models do achieve low prediction errors after seeing all 80 training scenarios.

We conjecture that the extremely asymmetric distribution of examples from different modes demonstrated here is characteristic of real-world environments. Many physical processes spend most of their time in steady states and only occasionally encounter boundary conditions. This reinforces the importance of online learning algorithms in such domains. Online learners are able to build accurate models of the most common modes quickly and use them to improve performance without having to wait for examples of uncommon modes. This is the behavior demonstrated by our system in figures 4.4 and 4.5. On the other hand, batch learners have to wait to collect a representative distribution of examples before they build any models, hurting

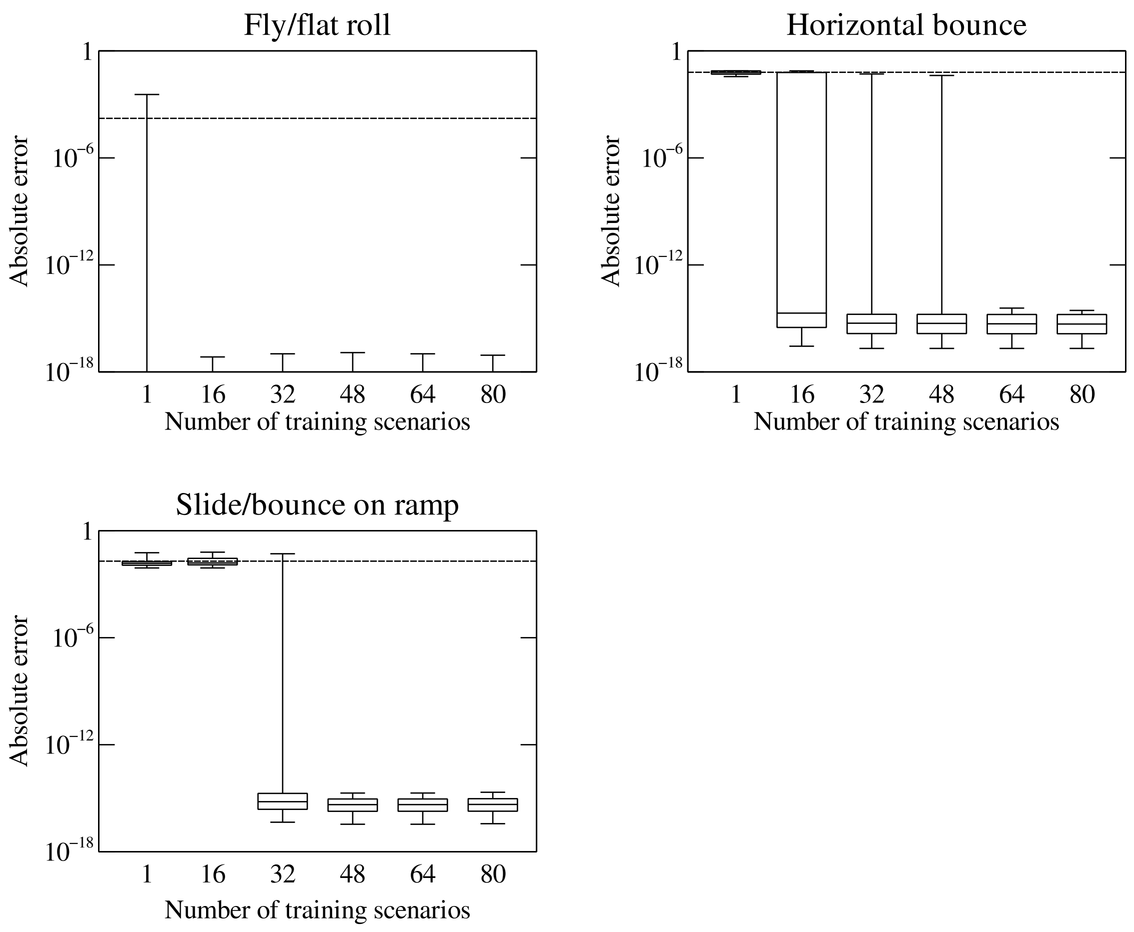


Figure 4.4: Prediction accuracy for x velocity, by mode

performance in the short term.

4.3.4 Classification accuracy

Prediction errors come from two sources. First, if the required modes have not been learned, or the mode functions are inaccurate, then the model will not be able to make the correct prediction. Second, even if the modes and mode functions are learned accurately, the classifier must still choose the correct mode to use when making a prediction. In this experiment, we have found that most of the prediction errors are due to the latter case.

We can separate these two sources of error by checking each test example for

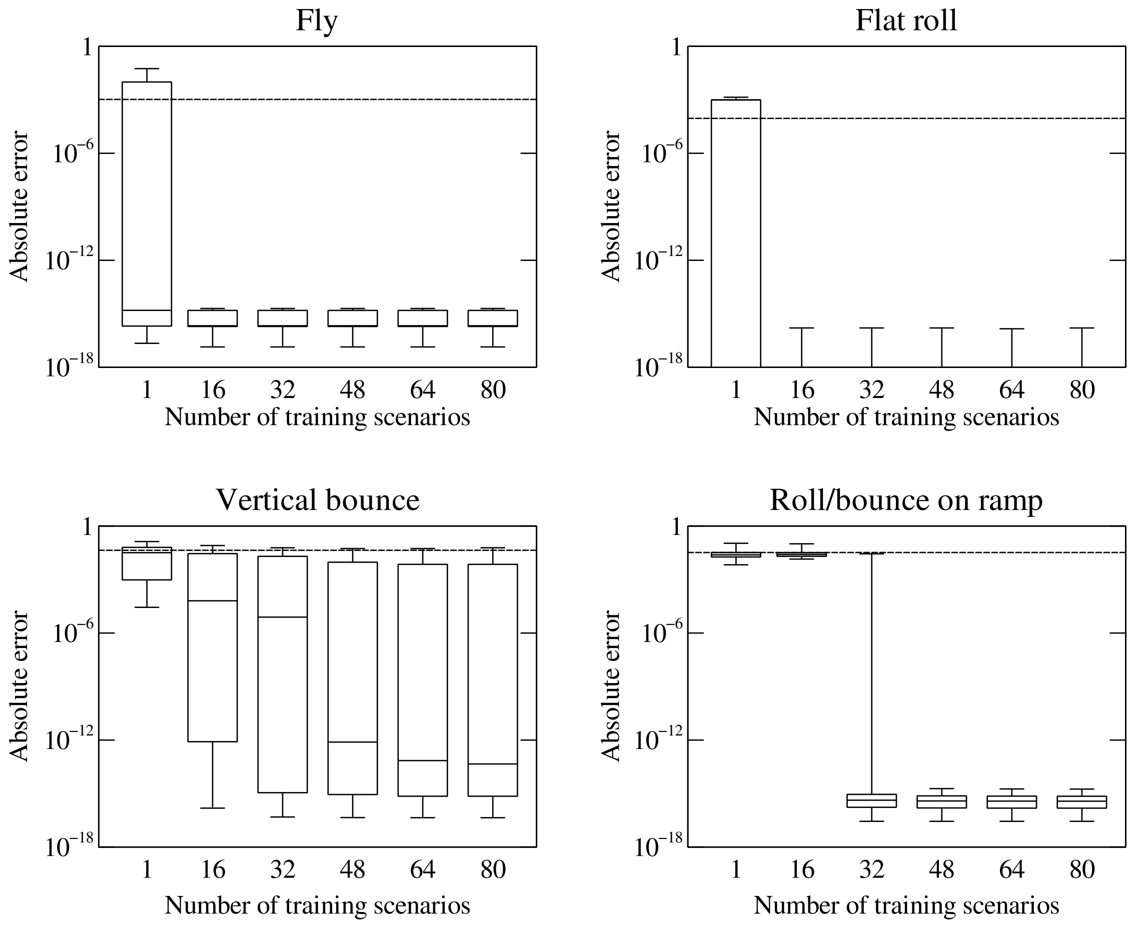


Figure 4.5: Prediction accuracy for y velocity, by mode

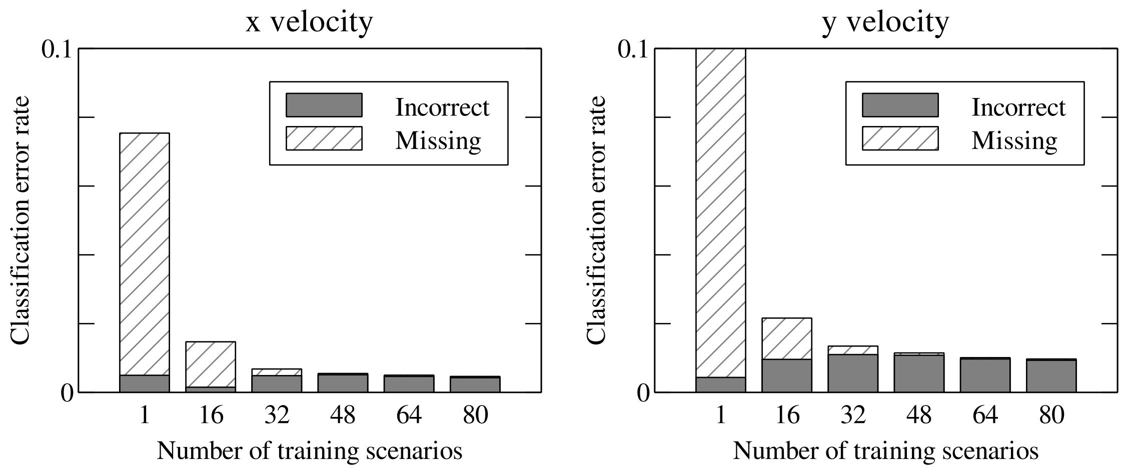


Figure 4.6: Classification error rates for x velocity and y velocity models.

Predicted mode	Correct mode 1	2	3	Total
1	-	12	1	13
2	26	-	17	43
3	63	1	-	64
Total errors	89	13	18	120
Num. Examples	23495	318	187	24000
Error rate	.000	.041	.096	.005

Table 4.7: Number of classification errors made by x velocity model for each pair of predicted/correct modes.

the prediction error that would result from using each possible mode. For each test example, we consider the mode that results in the lowest error to be the correct mode for that example. We then compare the correct modes obtained in this way with the modes chosen by the classifier to obtain classification error rates.

The classification error rates for both models are shown in figure 4.6. The striped bars in the figure indicate classification errors due to the correct mode not having been learned. The gray bars indicate classification errors where the correct mode has been learned but the classifier selected a different mode. The first bar for y velocity reaches a value of 0.2. This was not shown to maintain a reasonable scaling for the later bars. Note that classification errors after all 80 training scenarios have been seen is still at 2%.

Predicted mode	Correct mode 1	2	3	4	Total
1	-	69	0	0	69
2	6	-	91	0	97
3	4	0	-	1	5
4	75	12	1	-	88
Total errors	85	81	92	1	259
Num. examples	13530	9768	515	187	24000
Error rate	.006	.008	.179	.005	.011

Table 4.8: Number of classification errors made by y velocity model for each pair of predicted/correct modes.

Tables 4.7 and 4.8 show the types of classification errors being made by the x and y velocity models, respectively. The entry (i, j) in the table represents the number of times the model classifier chose mode i when the correct mode is j, where i is the row of the entry and j is the column of the entry. Note that these are results from a single model after being trained on all scenarios. There are small variations across other models learned with different random seeds and configuration orders. The most interesting feature to note is the high error rate for mode 3 (bouncing on horizontal surface) of the y velocity model. This explains why the prediction error for that mode is the highest amongst all modes, as shown in figure 4.5.

We can also see that the classifier frequently confuses mode 3 for mode 2. This is consistent with the relatively high training error of the binary classifier responsible for distinguishing between modes 2 and 3, as seen in table 4.5. Since mode 2 describes the ball sliding on a horizontal surface and mode 3 describes the ball bouncing off the same type of surface, no spatial relationship can distinguish between them. To lower the classification error between modes 2 and 3, the system must learn a numeric classifier that tests whether the y velocity of the ball in the current time step is zero (indicating it will slide) or negative (indicating it will bounce). However, the numeric classifier was not learned here because the false negative rate for the binary classifier did not exceed the threshold put in place to guard against overfitting the training

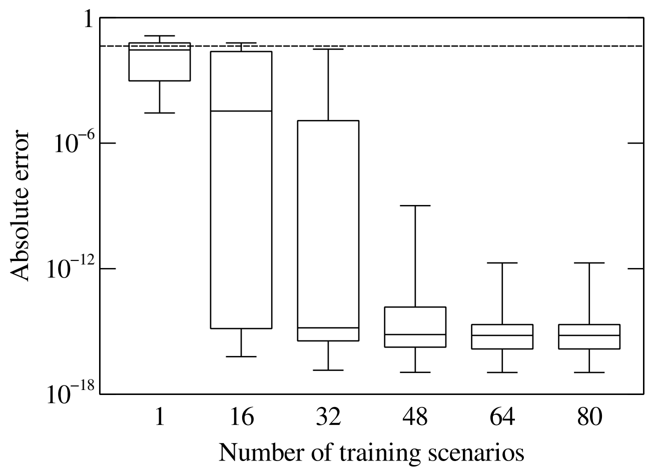


Figure 4.7: y velocity prediction error for bouncing on horizontal surface with perfect classifier.

data, as discussed in section 3.4.4. If the training examples had been distributed differently such that the system was trained on more examples of the ball bouncing, the false negative rate would have exceeded the threshold and the numeric classifier would have been learned. This is a shortcoming of our system that needs to be addressed in future work.

Finally, we can confirm that most of the prediction error is due to classification errors by looking at prediction accuracy if the correct mode is always chosen. Figure 4.7 plots prediction error for the y velocity model on test examples exhibiting mode 3 if the classifier always chose the correct mode. The high error up to 32 training scenarios is due to the correct mode not having been learned, but afterwards the error drops to essentially zero. This can be compared to figure 4.5 where the median error approaches zero after 32 training scenarios but the third quartile never improves.

4.4 Experiment 2

In the next experiment, we introduce friction into the simulation and also change some other simulation parameters. Friction constants for objects other than the ball are set at 0.2, elasticity constants have been changed to 0.95, and the ramp's incline has been changed to 45°. The presence of friction requires the system to learn additional modes to describe the ball's behavior completely. Specifically, the ball's x velocity model now requires a different mode for each direction it slides in. This is because the Coulomb friction model imposes a constant amount of deceleration in the direction opposite to the ball's direction of movement. If the ball is sliding to the right (positive x direction), a negative constant is added to its x velocity at each time step. If the ball is sliding to the left (negative x direction), a positive constant is added to its x velocity at each time step.

As will be discussed in more detail, the main consequence of introducing friction to the environment was an increase in the number of different modes the environment exhibited. A consequence of this increase is the number of training examples belonging to each mode decreases for a fixed size training set. In several cases, this resulted in the model not being able to learn certain modes whose examples occur infrequently during simulation, such as certain types of bounces. To account for this, we increased the training set size by repeating the same 40 block training set four times with different random seeds instead of only two times like we did in the frictionless experiments. This results in a total of $40 \times 4 \times 200 = 32000$ training examples in each training batch. We still use 10 reorderings and 5 sets of random seeds for a total of 50 training batches.

4.4.1 Learned modes

Table 4.9 shows the modes learned in the majority of training batches. For the x velocity model, adding friction to environment splits the behavior that was previously covered by a single mode (mode 1 in table 4.2) into three modes. The mode where x velocity remains constant (mode 3) now only describes the behavior of the ball when not contacting any surface. The interactions between the ball and horizontal surfaces, including both sliding and bouncing, are covered by modes 1 and 2, which we discuss

#	**Behavior**	**Num. examples**	**Mode function**
x velocity model			
0	Noise	65	-
1	Slide/bounce left on horizontal surface	6477	$v'_x = v_x - 0.39v_y + 1.96 \times 10^{-4}$
2	Slide/bounce right on horizontal surface	6094	$v'_x = v_x + 0.39v_y - 1.96 \times 10^{-4}$
3	Fly	18882	$v'_x = v_x$
4	Bounce on vert. surf.	347	$v'_x = -0.95v_x$
5	Halt from friction	93	$v'_x = 0$
6	Bounce/slide on ramp	42	$v'_x = -0.17v_x - 1.17v_y + 5.88 \times 10^{-4}$
y velocity model			
0	Noise	64	-
1	Fly (in gravity)	17151	$v'_y = v_y - 9.8 \times 10^{-4}$
2	Slide on horiz. surf.	13639	$v'_y = 0$
3	Bounce on horiz. surf.	308	$v'_y = -0.95v_y$
4	Bounce on vert. surf.	437	$v'_y = 0.39v_x + v_y - 9.8 \times 10^{-4}$
5	Bounce on vert. surf.	359	$v'_y = -0.39v_x + v_y - 9.8 \times 10^{-4}$
6	Slide/bounce on ramp	42	$v'_y = -0.78v_x + 0.22v_y - 5.88 \times 10^{-4}$

Table 4.9: Modes learned for x and y velocities of ball in environments with 0.2 friction, 0.95 elasticity, and 45° ramps.

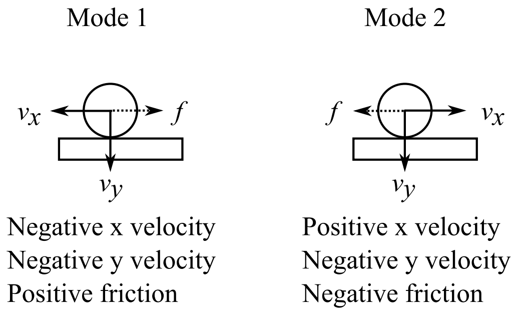


Figure 4.8: Possible cases of the ball bouncing on a horizontal surface.

in detail.

The functions for modes 1 and 2 in the x velocity model each cover two major cases - sliding and bouncing. In the sliding case, the y velocity of the ball (v_y) is zero, so the second term in the function drops out, leaving $v'_x = v_x \pm 1.96 \times 10^{-4}$. The constant 1.96×10^{-4} is due to a constant damping of the x velocity due to Coulomb friction, in the opposite direction of the ball's motion. Therefore, the constant is positive when the ball is moving left (negative x velocity) and negative when the ball is moving right (positive x velocity). The exact value is a combination of the friction constants of the interacting objects and a constant downard normal force that in turn depends on the ball's mass and the gravity constant.

In the bouncing case, there is an additional downward force proportional to the ball's y velocity that contributes to frictional drag. This is represented by the $\pm 0.39 v_y$ term in the functions for modes 1 and 2. Figure 4.8 illustrates why the v_y coefficient is negative in mode 1 and positive in mode 2. Because y velocity is alway negative when bouncing on a horizontal surface (the ball never bounces against the ceiling during the experiments), the sign of the coefficient is always the opposite of the sign of the actual frictional force.

Mode 4 is the same as the horizontal bounce mode in the frictionless surface since friction doesn't affect x velocity in these cases. However, the coefficient is -0.95 instead of -0.9 because of the change in elasticity constant. Mode 5 describes an edge case that occurs just as the ball comes to a halt due to friction. Mode 6 describes the behavior of the ball as it bounces or slides against the inclined surface of the ramp. Its coefficients have changed from the ramp mode in experiment 1 due to the change in ramp angle from 30° to 45°.

The y velocity model has also gained more modes. The first three modes are the same as in the frictionless case, except the mode for bouncing against a horizontal surface has a different constant due to the changed elasticity constant. Modes 4 and 5 are describe the motion of the ball as it bounces against a vertical surface such as the walls or side of the box as it is flying through the air. Like modes 1 and 2 in the x velocity model, the surface friction of the bounce damps the y velocity of the ball proportional to the normal force against the surface, which is in turn proportional to the x velocity of the ball. Unlike the bouncing cases x velocity model, where y velocity was always negative, the x velocity of the ball can be positive or negative during these bounces, in addition to the y velocity being positive or negative. Therefore, there are four cases to consider in explaining the sign of the v_x term, as illustrated in figure 4.9.

In configurations A and B in the figure, the sign of the x velocity is the same as the sign of the frictional force, so they are covered by mode 4, where the coefficient is positive. In configurations C and D, the sign of the x velocity is the opposite of the sign of the frictional force, so they are covered by mode 5, where the coefficient is negative.

4.4.2 Learned classifier

Due to the larger number of modes, the number of binary classifiers learned to distinguish between modes has also increased. The complete classifiers are shown in tables 4.10, 4.11, 4.12, 4.13, 4.14. We will not go into detail about the meaning of each binary classifier, as the analysis is similar to experiment 1. The major difference

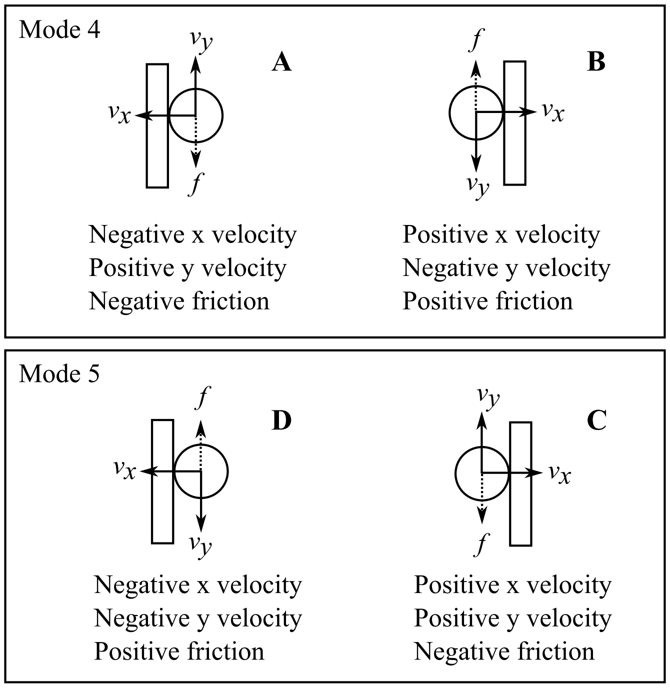


Figure 4.9: Four possible collision configurations.

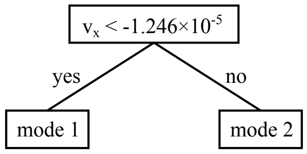


Figure 4.10: Decision tree distinguishing between modes 1 and 2 of the x velocity model.

we want to discuss is the presence of numeric classifiers.

The presence of friction has created pairs of modes that can only be distinguished by the sign of the ball's velocity. For example, modes 1 and 2 describe sliding or bouncing left and right on a horizontal surface. The only property that can distinguish examples in these two modes is whether the ball's x velocity is negative or positive. Since the system doesn't have a relation that tests this property, it must learn a numeric classifier. Table 4.10 shows that the one clause learned for the binary classifier for modes 1 and 2 only correctly classifies five examples, while the rest is left to the numeric classifier. In fact, the single learned clause is probably incorrect, and may cause misclassifications on test examples.

Clause	**Num. correct**	**Num. incorr.**	**Numeric class.?**
Mode 1 vs. 2 (train perf. 1)			
$\neg intersect(A,*) \wedge box(B)$	5	0	No
Negative	6094	6472	Yes
Mode 1 vs. 3 (Negated) (train perf. .9571)			
$\neg intersect(A,*) \wedge y\text{–}aligned(A,B)$	11083	0	No
$\neg x\text{–}aligned(A,*)$	564	0	No
$ramp(B)$	77	0	No
$\neg intersect(A,*) \wedge box(B)$	1161	3	No
$\neg intersect(A,*) \wedge floor(B)$	4947	34	No
Negative	6440	1050	No
Mode 1 vs. 4 (train perf. .9996)			
$\neg y\text{–}aligned(A,*)$	6469	0	No
$\neg on\text{–}top(A,*) \wedge floor(B)$	5	0	No
Negative	347	3	No
Mode 1 vs. 5 (train perf. .9965)			
$box(B)$	481	0	No
$x\text{–}greater\text{–}than(B,A)$	3	0	No
$on\text{–}top(A,B)$	5858	23	No
$\neg intersect(A,*)$	130	0	No
$y\text{–}aligned(A,B)$	5	0	No
Negative	70	0	No
Mode 1 vs. 6 (train perf. 1.0)			
$\neg y\text{–}aligned(A,*)$	6469	0	No
$x\text{–}greater\text{–}than(B,A)$	3	0	No
$\neg on\text{–}top(A,*) \wedge floor(B)$	5	0	No
Negative	42	0	No
Mode 2 vs. 3 (Negated) (train perf. 0.9573)			
$\neg intersect(A,*) \wedge y\text{–}aligned(A,B)$	11083	0	No
$\neg x\text{–}aligned(A,*)$	564	0	No
$\neg intersect(A,*) \wedge \neg floor(*)$	1236	0	No
$\neg intersect(A,*)$	4947	14	No
Negative	6080	1052	No
Mode 2 vs. 4 (train perf. 0.9995)			
$\neg y\text{–}aligned(A,*)$	6071	0	No
$\neg on\text{–}top(A,*) \wedge floor(B)$	20	0	No
Negative	347	3	No

Table 4.10: Clauses learned for x velocity classifier, part 1

Clause	**Num. correct**	**Num. incorr.**	**Numeric class.?**
Mode 2 vs. 5 (train perf. 0.9934)			
$box(B)$	291	0	No
$x\text{–}greater\text{–}than(A, B)$	2	0	No
$on\text{–}top(A, B)$	5739	20	No
$\neg on\text{–}top(A, *) \wedge y\text{–}greater\text{–}than(A, B)$	42	1	No
Negative	72	20	No
Mode 2 vs. 6 (train perf. 1.0)			
$\neg y\text{–}aligned(A, *)$	6071	0	No
$\neg on\text{–}top(A, *) \wedge floor(B)$	20	0	No
$x\text{–}greater\text{–}than(A, B)$	3	0	No
Negative	42	0	No
Mode 3 vs. 4 (train perf. 0.9995)			
$\neg intersect(A, *)$	17820	0	No
$\neg y\text{–}aligned(A, *)$	1048	0	No
$\neg y\text{–}greater\text{–}than(A, *) \wedge x\text{–}aligned(A, B)$	4	0	No
Negative	347	10	No
Mode 3 vs. 5 (train perf. 0.9972)			
$\neg intersect(A, *) \wedge y\text{–}aligned(A, B)$	11083	0	No
$\neg intersect(A, *)$	6737	0	No
Negative	93	1062	Yes
Mode 3 vs. 6 (train perf. 0.9998)			
$\neg intersect(A, *)$	17820	0	No
$\neg y\text{–}aligned(A, *)$	1048	0	No
$\neg x\text{–}aligned(A, *)$	10	0	No
Negative	42	4	No
Mode 4 vs. 5 (train perf. 1.0)			
$\neg floor(*)$	123	0	No
$y\text{–}aligned(A, B) \wedge on\text{–}top(A, C)$	224	0	No
Negative	93	0	No
Mode 4 vs. 6 (train perf. 0.9923)			
$x\text{–}greater\text{–}than(B, A)$	186	0	No
$x\text{–}greater\text{–}than(A, B)$	140	0	No
$\neg ramp(*)$	18	0	No
Negative	42	3	No
Mode 5 vs. 6 (train perf. 1.0)			
$\neg y\text{–}aligned(A, *)$	85	0	No
$\neg on\text{–}top(A, *) \wedge floor(B)$	8	0	No
Negative	42	0	No

Table 4.11: Clauses learned for x velocity classifier, part 2

Clause	**Num. correct**	**Num. incorr.**	**Numeric class.?**
Mode 1 vs. 2 (train perf. 0.9969)			
$\neg intersect(A,*) \wedge y\text{–}aligned(A,B)$	11016	0	No
$\neg intersect(A,*) \wedge \neg x\text{–}aligned(A,*)$	553	0	No
$\neg intersect(A,*) \wedge ramp(B)$	75	0	No
$\neg intersect(A,*) \wedge box(B)$	1160	1	No
$\neg intersect(A,*) \wedge floor(B)$	4306	52	No
Negative	13586	41	No
Mode 1 vs. 3 (train perf. 0.9982)			
$\neg intersect(A,*)$	17110	0	No
$\neg x\text{–}aligned(A,*)$	10	0	No
Negative	308	31	No
Mode 1 vs. 4 (train perf. 0.9935)			
$\neg intersect(A,*) \wedge ramp(B)$	5594	0	No
$x\text{–}greater\text{–}than(A,B) \wedge box(B)$	930	1	No
$\neg x\text{–}aligned(A,*) \wedge y\text{–}greater\text{–}than(A,B)$	270	0	No
$\neg intersect(A,*) \wedge x\text{–}greater\text{–}than(B,A) \wedge wall(B)$	1723	1	No
$\neg intersect(A,*) \wedge x\text{–}greater\text{–}than(A,B)$	983	4	No
$\neg intersect(A,*) \wedge box(B)$	3304	7	No
$y\text{–}greater\text{–}than(A,B) \wedge intersect(A,C)$	28	0	No
$y\text{–}greater\text{–}than(A,B)$	4306	88	No
Negative	337	13	No
Mode 1 vs. 5 (train perf. 0.9953)			
$ramp(B)$	5598	0	No
$\neg x\text{–}aligned(A,*) \wedge y\text{–}greater\text{–}than(A,B)$	547	0	No
$\neg intersect(A,*) \wedge x\text{–}greater\text{–}than(A,B)$	1636	1	No
$\neg intersect(A,*) \wedge box(B)$	3304	0	No
$wall(B) \wedge \neg intersect(A,*)$	1723	3	No
$y\text{–}greater\text{–}than(A,B) \wedge intersect(A,C)$	28	0	No
$y\text{–}greater\text{–}than(A,B)$	4306	70	No
Negative	285	9	No
Mode 1 vs. 6 (train perf. 0.9998)			
$\neg intersect(A,*)$	17110	0	No
$\neg y\text{–}aligned(A,*)$	28	0	No
$\neg x\text{–}aligned(A,*)$	10	0	No
Negative	42	3	No

Table 4.12: Clauses learned for y velocity classifier, part 1

Clause	**Num. correct**	**Num. incorr.**	**Numeric class.?**
Mode 2 vs. 3 (train perf. 0.9949)			
$\neg intersect(A, *)$	195	0	No
$x\text{–}greater\text{–}than(B, A)$	133	0	No
$x\text{–}greater\text{–}than(A, B)$	109	0	No
$\neg y\text{–}aligned(A, *) \wedge \neg on\text{–}top(A, *)$	462	0	No
$\neg y\text{–}aligned(A, *) \wedge floor(B)$	12584	28	No
Negative	280	156	Yes
Mode 2 vs. 4 (train perf. 0.9943)			
$on\text{–}top(A, B)$	12960	0	No
$intersect(A, B) \wedge y\text{–}greater\text{–}than(A, C)$	464	0	No
Negative	437	215	Yes
Mode 2 vs. 5 (train perf. 0.9856)			
$on\text{–}top(A, B)$	12960	0	No
$intersect(A, B) \wedge y\text{–}greater\text{–}than(A, C)$	464	0	No
Negative	359	215	Yes
Mode 2 vs. 6 (train perf. 0.9999)			
$\neg y\text{–}aligned(A, *)$	13388	0	No
$x\text{–}greater\text{–}than(B, A)$	133	0	No
$x\text{–}greater\text{–}than(A, B)$	109	0	No
$box(B)$	7	0	No
Negative	42	2	No
Mode 3 vs. 4 (train perf. 1.0000)			
$on\text{–}top(A, B)$	274	0	No
$intersect(A, B) \wedge floor(B)$	34	0	No
Negative	437	0	No
Mode 3 vs. 5 (train perf. 1.0000)			
$on\text{–}top(A, B)$	274	0	No
$intersect(A, B) \wedge floor(B)$	34	0	No
Negative	359	0	No
Mode 3 vs. 6 (train perf. 0.9914)			
$\neg y\text{–}aligned(A, *)$	271	0	No
$\neg on\text{–}top(A, *) \wedge floor(B)$	34	0	No
Negative	42	3	No
Mode 4 vs. 5 (train perf. 0.8445)			
Negative	359	437	Yes

Table 4.13: Clauses learned for y velocity classifier, part 2

Clause	**Num. correct**	**Num. incorr.**	**Numeric class.?**
Mode 4 vs. 6 (train perf. 0.9979)			
$\neg intersect(A, *)$	388	0	No
$\neg x\text{–}aligned(A, *)$	43	0	No
$box(B)$	5	0	No
Negative	42	1	No
Mode 5 vs. 6 (train perf. 1.0000)			
$\neg intersect(A, *)$	308	0	No
$\neg x\text{–}aligned(A, *)$	47	0	No
$\neg ramp(*)$	4	0	No
Negative	42	0	No

Table 4.14: Clauses learned for y velocity classifier, part 3

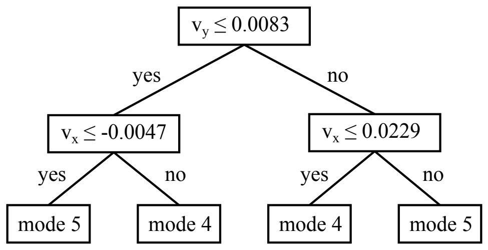


Figure 4.11: Decision tree distinguishing between modes 4 and 5 of the y velocity model.

As previously mentioned, we use a decision tree learner to learn numeric classifiers. The decision tree for modes 1 and 2 of the x velocity model is shown in figure 4.10. The decision tree has learned correctly that a negative x velocity suggests mode 1, while a positive x velocity suggests mode 2, as was illustrated in figure 4.8. The root node of the tree checks v_x against a small negative value instead of 0 because the decision tree learning procedure sets the tested value midway between two data points from different modes. Figure 4.11 shows a larger tree learned to distinguish between the four vertical friction cases for the y velocity model illustrated in figure 4.9.

The binary classifier for x velocity modes 3 and 5 also has a numeric classifier. But these modes are not distinguished by friction. Instead, notice that once the ball's x velocity reaches 0 and it stops, its behavior can be described by either mode 3 ($v'_x = v_x$) or mode 5 ($v'_x = 0$) equally accurately, so the algorithm arbitrarily assigns each such example to either mode. The result is that the two modes cannot be correctly distinguished by clauses, so a numeric classifier is learned. Even though the numeric classifier gives a lower training error, it is most likely overfitting the data and will not perform well on test examples. The root of the problem is the unprincipled way in which our algorithm deals with examples that can be assigned to multiple modes. This is a weakness of the algorithm that should be addressed in future work.

4.4.3 Prediction accuracy

We now present results for the overall prediction accuracy of the learned models. These results were obtained using the same testing procedure as in the previous experiment, except the number of training examples was increased, as discussed at the beginning of this section.

As in experiment 1, we show the overall prediction accuracy of the models (figure

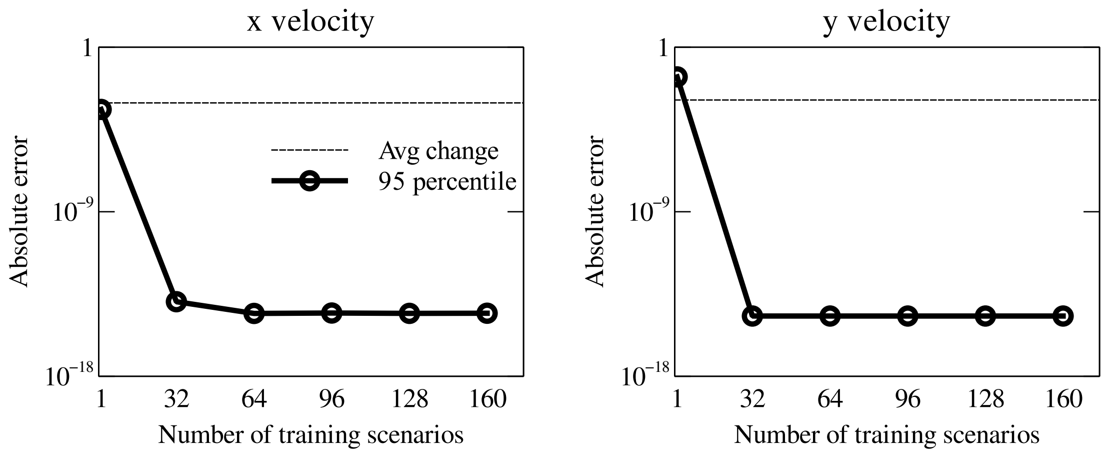


Figure 4.12: Prediction accuracy for x velocity and y velocity models

4.12), as well as accuracy within individual modes (figures 4.13 and 4.14). Also as in experiment 1, we see that most modes are learned very quickly, but a few modes are difficult to learn. Specifically, predictions for the halt mode of the x velocity model and the bouncing on horizontal surface mode in the y velocity model have consistently high errors throughout the training sequence. Again, these errors are mostly due to high misclassification rates for those modes, as we will discuss in the next section.

4.4.4 Classification accuracy

Here we analyze the performance of the learned classifiers in the same fashion as in experiment 1. We first show overall classification error rates for x and y velocity models in figure 4.15. The error rates are significantly higher than those obtained in experiment 1, which is expected due to the increased complexity of the environment and the increase in number of modes from 3 to 6. After seeing all training examples, the x velocity classifier achieves an error rate of 3.6%, while the y velocity classifier achieves an error rate of 1.3%.

Tables 4.15 and 4.16 show how classification errors are distributed across the individual modes. For x velocity, the highest absolute number of errors result from classifying mode 3 examples as mode 2 examples, while the highest percentage of

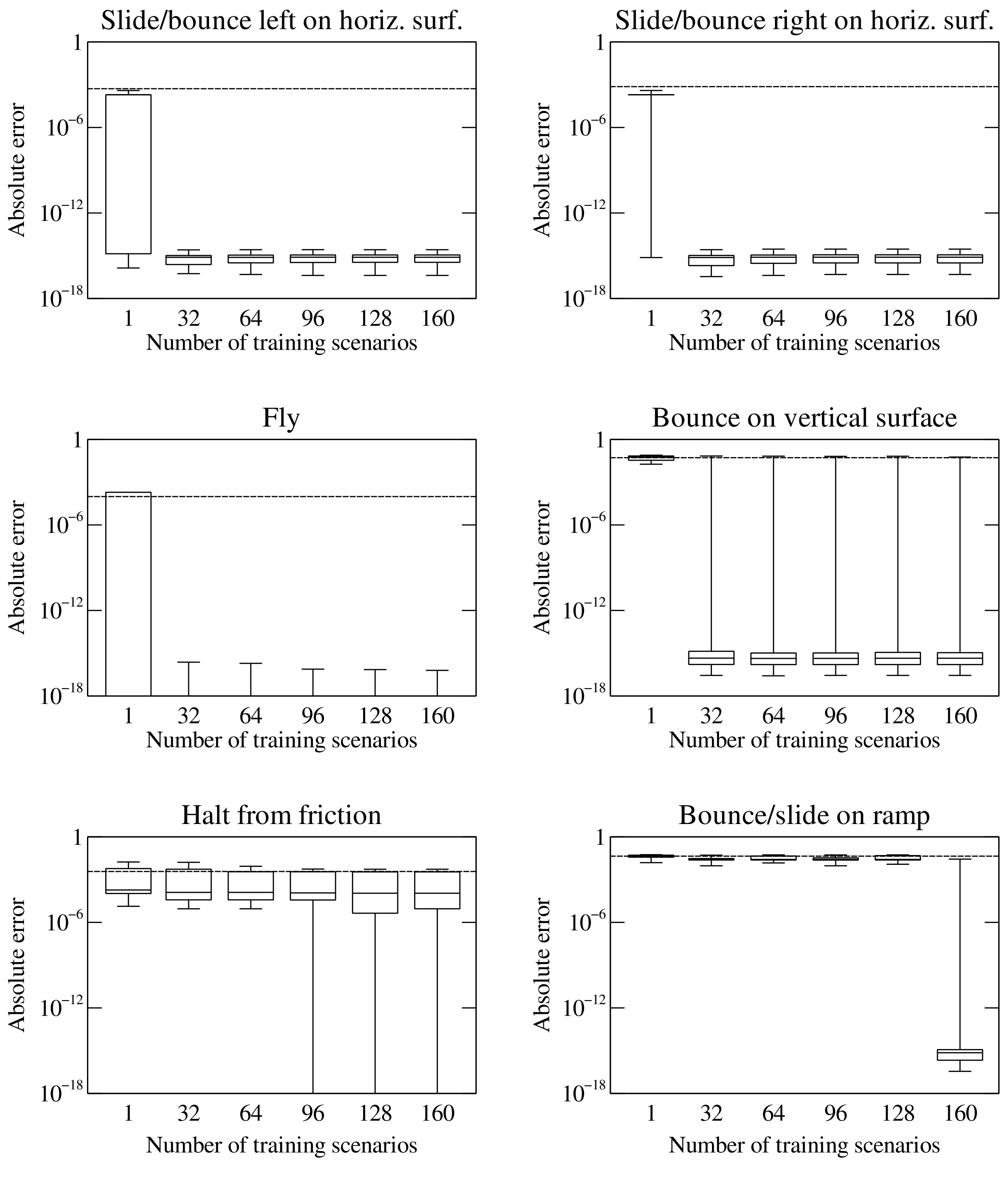


Figure 4.13: Prediction accuracy for x velocity, by mode

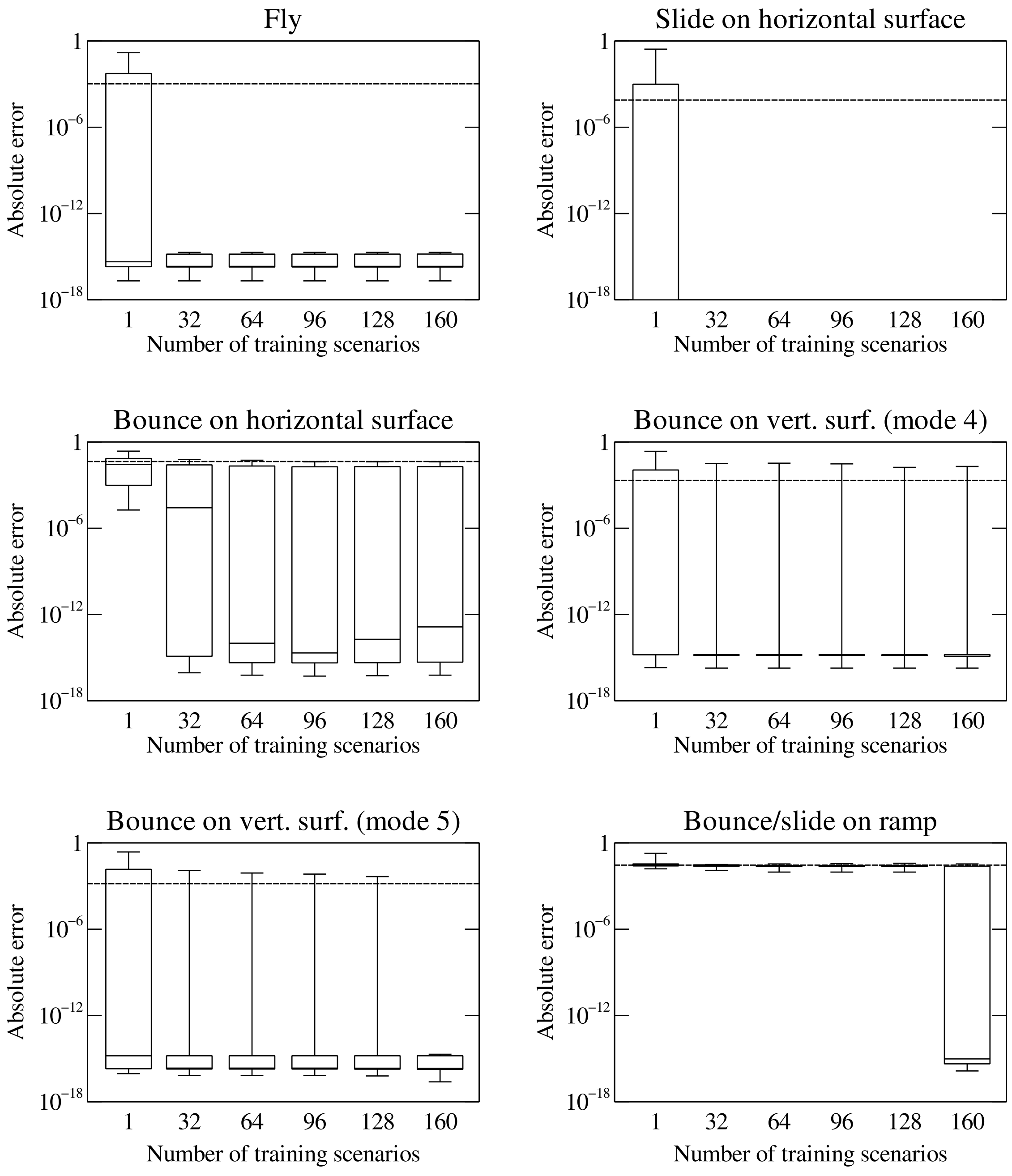


Figure 4.14: Prediction accuracy for y velocity, by mode

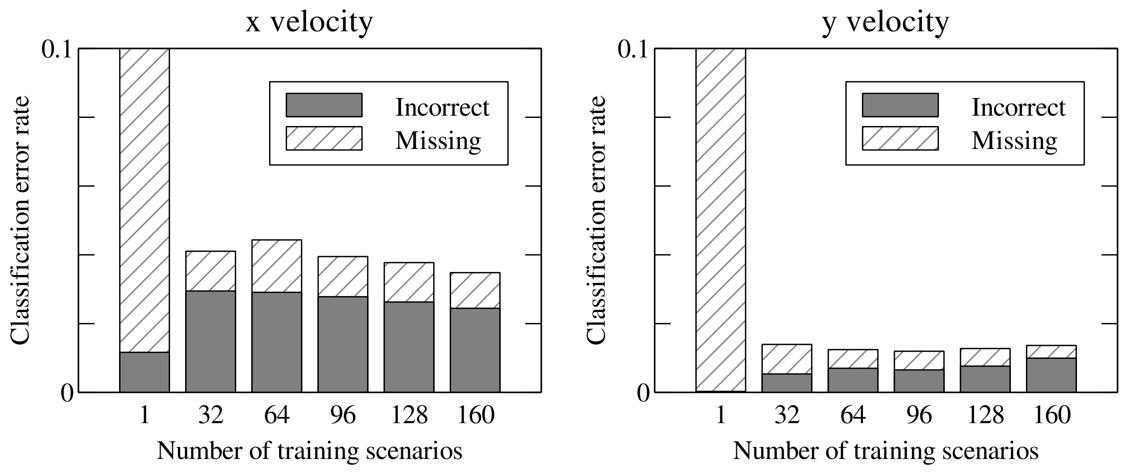


Figure 4.15: Classification error rates for x velocity and y velocity models.

errors result from classifying mode 5 examples as mode 1 or mode 2 examples. The latter result explains why the prediction accuracy for mode 5 (the halting mode) is significantly higher than the other modes, as seen in figure 4.13. We hypothesize that this high classification error rate is due to the asymmetry in the number of examples of each mode the learning algorithm experiences. Because the algorithm sees several thousand examples each of modes 1 and 2 but less than 100 examples of mode 5, a small number of misclassifications of mode 5 are ignored in favor of obtaining short clauses that cover large numbers of mode 1 and 2 examples correctly. The clauses shown in tables 4.10 and 4.11 for mode 1 versus 5 and mode 2 versus 5 support this hypothesis. In each of those binary classifiers, the clause that covers the most positive examples is $\mathit{on{-}top}(A, B)$. This clause simply says that whenever the ball is on a surface, classify it as sliding under friction rather than halting. This is clearly overgeneral, as many examples of halting occur when the ball is on top of a surface.[1] If an equal number of halting versus sliding examples were presented to our algorithm, we expect that such an overgeneral clause would result in an unacceptable

[1]Actually, all the examples of halting occur when the ball is on a surface. However, in many of the cases the ball has penetrated into the surface due to imperfections in the physics simulation, to the degree where our definition of *on–top* no longer holds. This is an example of the algorithm overfitting artifacts in the physics simulator.

Predicted mode	Correct mode 1	2	3	4	5	6	Total
1	-	0	3	0	20	0	23
2	0	-	620	0	43	0	663
3	32	16	-	0	0	0	48
4	5	2	0	-	0	0	7
5	0	25	0	0	-	0	25
6	0	0	3	11	0	-	14
Total errors	37	43	626	11	63	0	780
Num. Examples	4746	4976	13936	199	67	30	23954
Error rate	0.008	0.009	0.045	.055	0.940	0	0.033

Table 4.15: Number of classification errors made by x velocity model for each pair of predicted/correct modes.

Predicted mode	Correct mode 1	2	3	4	5	6	Total
1	-	48	0	0	0	0	48
2	12	-	138	1	0	0	151
3	0	0	-	0	0	0	0
4	0	1	0	-	1	0	2
5	1	10	0	5	-	0	16
6	15	69	138	7	5	-	234
Total errors	15	69	138	7	5	0	234
Num. Examples	13325	10337	210	23	30	30	23955
Error rate	0.001	0.007	0.657	.304	0.167	0	0.010

Table 4.16: Number of classification errors made by y velocity model for each pair of predicted/correct modes.

false positive rate and be rejected. The correct distinguishing feature should be a decision tree that tests whether the x velocity is below a threshold such that friction will cause it to go to 0 in the next time step.

4.5 Experiment 3

In this experiment, we train the model on two different environments, with all objects having friction coefficients of 0.1 or 0.2. The training procedure is the same as in experiment 2, except with two different sets of environments. The primary purpose of this experiment is to show that our system can handle a larger number of modes than in the previous experiments.

In order to distinguish between objects with different friction coefficients, the coefficient is encoded in the object types. Therefore, a ramp with friction coefficient 0.1 has type *ramp-0.1*, whereas one with friction coefficient 0.2 has type *ramp-0.2*. The obvious alternative is to encode the friction coefficient as a property of the object, so that all ramps have type *ramp*, but have either 0.1 or 0.2 for their "friction coefficient" property. The problem with this alternative encoding is that it would introduce non-linear terms into the mode functions, such as $k \cdot v_x \cdot C_f$, where v_x is the ball's x velocity and C_f is another object's friction coefficient. By appending the friction coefficient to the object type, the mode functions will remain linear, and interactions between objects with different friction coefficients will result in distinct modes.

Table 4.17 shows the x velocity modes learned by the system. In addition to the modes from experiment 2, the system has learned an additional set of modes for the environments with 0.2 friction. This is the expected behavior. The model also learned two spurious modes (15 and 16). Figures 4.16 and 4.17 show the overall prediction error rate and mode classification error rate for the model when tested on environments with both friction coefficients. The testing procedure is the same as that of experiment 2. The plots show that classification error rate has increased slightly, as to be expected from the model having twice as many modes and an order of magnitude more binary classifiers. Overall, the system is able to accommodate the extra modes.

4.6 Noise Tolerance

As discussed in section 3.2.2, the expected amount of variance in the training examples due to noise is controlled by the parameter σ. Here, we will show how the system behaves under different settings of expected and real variance in the obser-

#	**Behavior**	**Num. examples**	**Mode function**
0	Noise	179	-
1	Slide right, 0.1 fric.	5253	$v'_x = v_x - 9.8 \times 10^{-5}$
2	Slide left, 0.1 fric.	4766	$v'_x = v_x + 9.8 \times 10^{-5}$
3	Slide right, 0.2 fric.	6343	$v'_x = v_x - 1.96 \times 10^{-4}$
4	Slide left, 0.2 fric.	6623	$v'_x = v_x + 1.96 \times 10^{-4}$
5	Fly	46031	$v'_x = v_x$
6	Bounce on vert. surf.	1191	$v'_x = -0.95v_x$
7	Halt from friction	7892	$v'_x = 0$
8	Bounce right, 0.1 fric.	4318	$v'_x = v_x + 0.195v_y - 9.8 \times 10^{-5}$
9	Bounce right, 0.2 fric.	2656	$v'_x = v_x + 0.39v_y - 1.96 \times 10^{-4}$
10	Bounce left, 0.1 fric.	6358	$v'_x = v_x - 0.195v_y + 9.8 \times 10^{-5}$
11	Bounce right, 0.2 fric.	4151	$v'_x = v_x - 0.39v_y + 1.96 \times 10^{-4}$
12	Ramp bounce, 0.1 fric.	79	$v'_x = -0.0725v_x - 1.0725v_y + 5.39 \times 10^{-4}$
13	Ramp bounce, 0.2 fric.	63	$v'_x = -0.17v_x - 1.17v_y + 5.88 \times 10^{-4}$
14	Ramp bounce, 0.1 fric.	50	$v'_x = -0.475v_x - 0.475v_y$
15	Ramp bounce, 0.1 fric.	47	$v'_x = 0.1225v_x - 0.8775v_y + 4.41 \times 10^{-4}$

Table 4.17: Modes learned for x velocity of ball in environments with 0.1/0.2 friction, 0.95 elasticity, and 45° ramps.

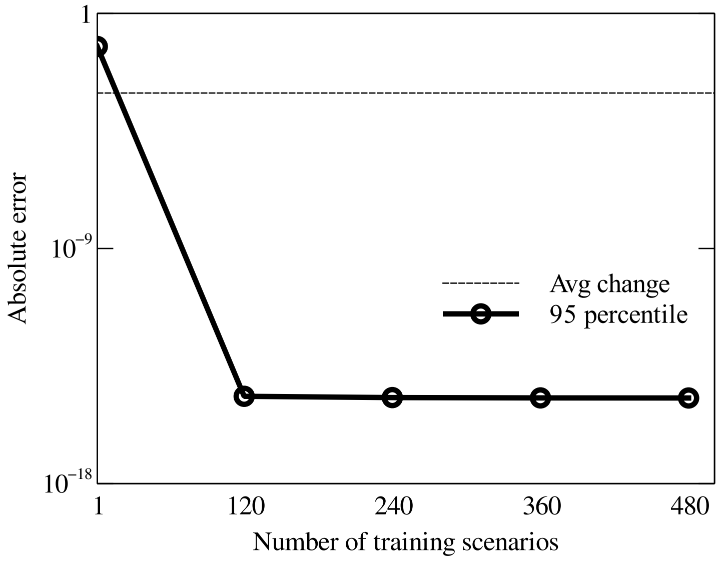


Figure 4.16: Prediction error for x velocity.

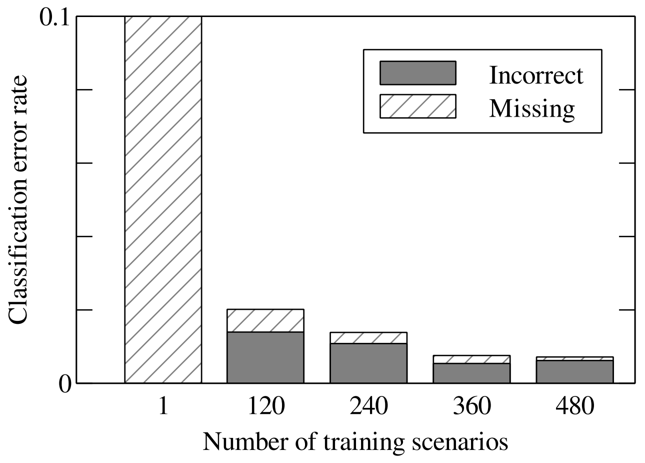


Figure 4.17: Classification error for x velocity.

vation noise. We use the frictionless ball-ramp-box domain in this experiment. We train and test the models in the same way as in section 4.3, but sweep the variance of the noise distribution used to corrupt the values in the state vector. The values swept over are 10^{-20}, 10^{-15}, 10^{-10}, 10^{-8}, and 10^{-5}. We also sweep σ at the same intervals. Model accuracy is only tested after all 80 training scenarios are presented. The experiment was repeated 15 times with different reorderings and random seeds. The result is shown in figure 4.18.

In the figure, the x axis marks the true variance of the environment noise. The y axis is different from the previous experiments. Instead of absolute error, the y axis now shows what we will call mode averaged median error. The mode averaged median error compensates for the uneven distribution of training examples amongst the modes by calculating the median error of each mode individually, and then taking their average. This way, all modes are represented equally regardless of their membership size. Each line corresponds to the median errors of a y velocity model obtained with one setting of σ.

Consider the lines representing $\sigma = 10^{-15}$ and $\sigma = 10^{-10}$. Their error rates are approximately the same until the real noise variance significantly exceeds the expected variance at an x value of 10^{-8}. Then the error rate of $\sigma = 10^{-15}$ becomes

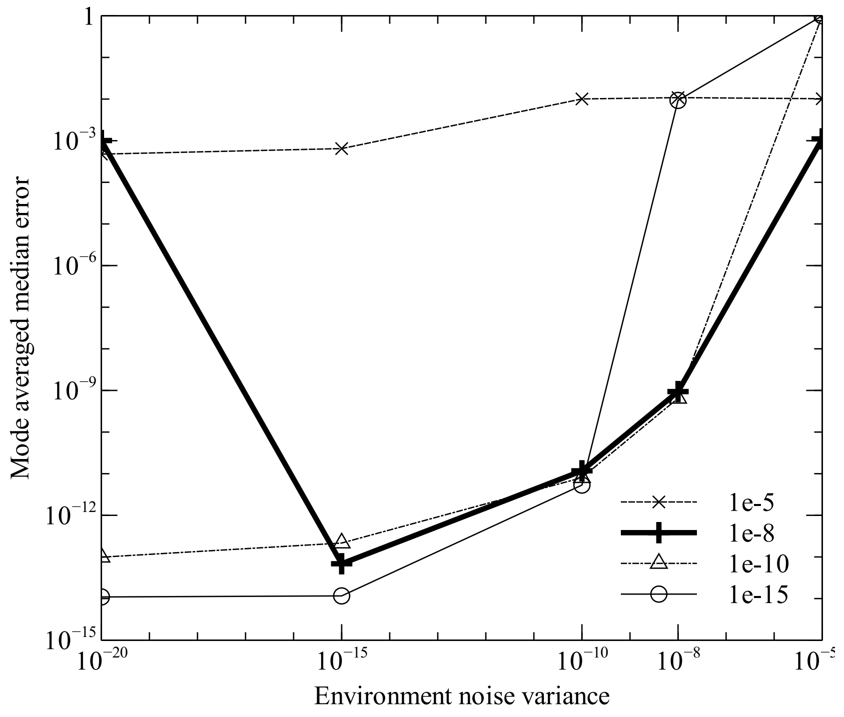


Figure 4.18: Model accuracy for different settings of environment and expected noise variance.

significantly higher than $\sigma = 10^{-10}$. Manual inspection of the learned models shows that $\sigma = 10^{-15}$ model does not learn the ramp bouncing mode correctly at this point, while the $\sigma = 10^{-10}$ model still learns it in most cases. Therefore, a σ setting that is significantly lower than the real environment noise variance disrupts mode learning. Note that the right-most plot point for these two lines is artificial. At an environment noise variance setting of 10^{-5}, these two models did not learn any modes at all and considered all training examples to be noise. This is arguably a desirable behavior, since any modes learned under a large amount of noise will probably not be accurate anyway.

On the other hand, the $\sigma = 10^{-5}$ line suggests that a sigma setting that is too high also hurts accuracy. In this case, the higher σ value results in a larger acceptable margin of error for a training example to be considered to belong to a mode, causing the algorithm to overfit the data and learn spurious modes that are over-general. In the worst case, the model learns a single mode that covers the majority of the training examples. This is essentially like not distinguishing between modes at all, and the prediction accuracy suffers because the single mode is an average over the behaviors of the real modes, and does not predict any one mode accurately.

The most interesting line is $\sigma = 10^{-8}$, as it has high error at both extremes of environment noise but low error in the middle. Inspection of the learned models shows that this setting always learns the flying and bouncing against a horizontal surface modes correctly, except when environment noise was at 10^{-5}. The large variation in accuracy is primarily due to whether the ramp bouncing mode was learned correctly. When environment noise is set to 10^{-20}, the algorithm overfit the data and learned spurious modes. At noise levels of 10^{-15}, 10^{-10}, and 10^{-8}, there was enough noise to prevent overfitting, and the algorithm was able to learn the correct model. Therefore, a small amount of environment noise actually helped to prevent overfitting. At a noise level of 10^{-5}, the algorithm learned some of the modes correctly, but also learns some spurious modes due to the high amount of real noise.

4.7 Comparison to other techniques

In this section, we compare the performance of our algorithm to other model learning techniques.

4.7.1 Prediction accuracy compared to LWR

We first compare the overall prediction accuracy of models learned by our algorithm with those learned by Locally Weighted Regression. The comparison was made in the environment described in experiment 2: the ball-box-ramp environment with friction. The LWR model was trained in the same way as our algorithm. Like in experiment 2, the data is aggregated over 50 different training batches generated from 10 different reorderings and 5 different random seed sets. We used 50 neighbors for each prediction of LWR, and weighed the neighbors with the kernel function d^{-3}, where d is the Euclidean distance of the neighbor from the point being predicted. Through informal experimentation, we found that increasing the number of neighbors decreases worst-case prediction errors but increases median prediction errors. This is because LWR smoothes the prediction over all neighbors without regard to different behavioral modes.

Recall that for each training and test example, we shift the positions of all objects in the environment by a random amount and in a random direction with respect to the coordinate origin. This was to prevent our algorithm from learning mode functions that condition on incidental invariances in the object coordinates. These random shifts are harmful to LWR because it uses the Euclidean distance between examples as a measure of similarity. With random shifts, examples that are otherwise identical would be considered very far apart, while examples that are different in configuration but shifted to the same origin would appear relatively more similar. In order to

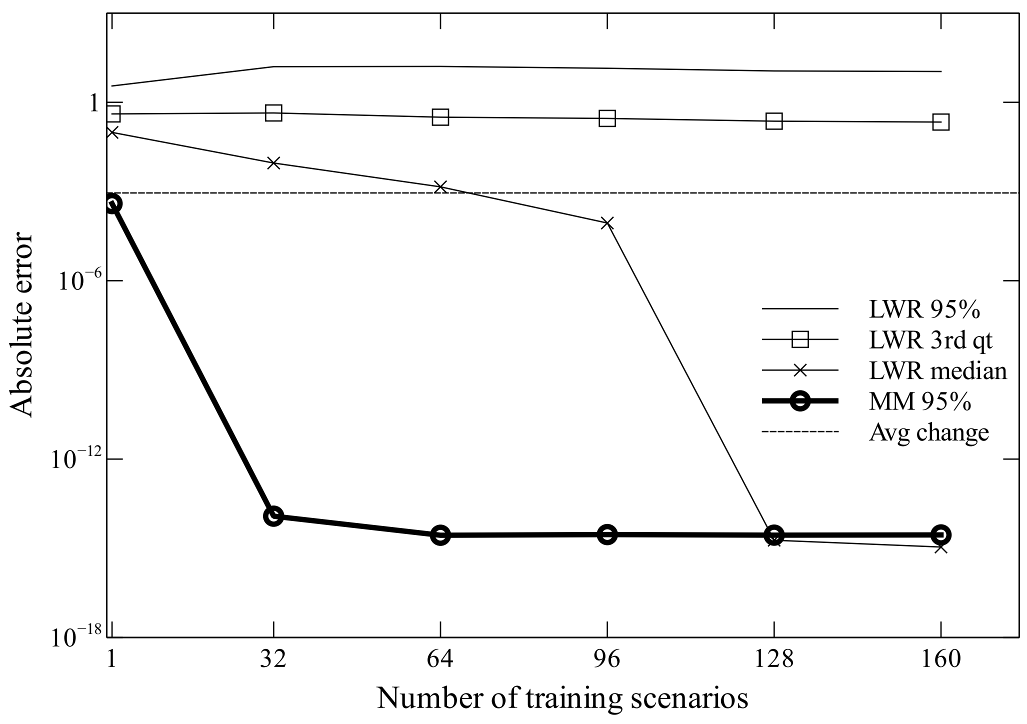


Figure 4.19: Comparison of prediction accuracy of our algorithm with LWR

make the distance metric agnostic to the random shifts and therefore improve the performance of LWR, we centered the position coordinates of each object on the target object, which is the ball in this case.

LWR's prediction accuracy and a comparison to our system is shown in figure 4.19. The thick line represents the 95^{th} percentile of our algorithm's accuracy for the x velocity model, which is reproduced from figure 4.12. The thin lines represent the 95^{th} percentile, 3^{rd} quartile, and median accuracies of the LWR model. The graph shows that LWR's prediction errors are significantly higher than our model's. The median error of LWR is much higher than our 95 percentile error for most of the training sequence, and the 3^{rd} quartile is never lower than the average magnitude of change. Although it's difficult to understand what LWR is doing, we hypothesize that its poor performance is due to the following reasons. First, LWR generalizes training examples to test examples poorly because it uses Euclidean distance as a similarity measure. Euclidean distance doesn't capture spatial relationships between objects, which is what really determines behavior in this domain. Therefore, LWR is generalizing from the wrong training examples to test examples.

Second, LWR smoothes over training examples close to each query point, regardless of the mode they exhibit. Training examples from flying, bouncing, and sliding modes all contribute to each prediction, making the prediction an average over these qualitatively different behaviors. This average prediction does not accurately predict any example from any of the modes. This phenomenon is reflected in the sudden decrease in median error at the 128 training scenarios mark. We believe this decrease occurs because there are enough training examples from the most common mode (ball flying) that they saturate the 50 closest neighbor slots used to predict each test example. Since the test examples are also predominantly from the most common mode, the median prediction error decreases dramatically. But prediction errors for test examples from all other modes are still poor, so the 3^{rd} quartile error does not decrease by the same magnitude. We also tried using 25 neighbors for each LWR prediction, and as expected this resulted in the median error dip occurring earlier in the training sequence. However, this also resulted in increased 3^{rd} quartile errors across the board.

We can see these effects amplified if we don't center the positions of objects around the ball in the training and test data. This is shown in figure 4.20. Centering object positions makes their position coordinates correlate with their distance from the ball, which correlates with important relationships such as intersection. Therefore, a lower Euclidean distance between two examples makes it more likely that the objects in the examples have the same relationship with the ball, so it becomes a more useful similarity metric. When objects are not centered, the Euclidean distance much less correlated with relationships with the ball, and performance of LWR suffers.

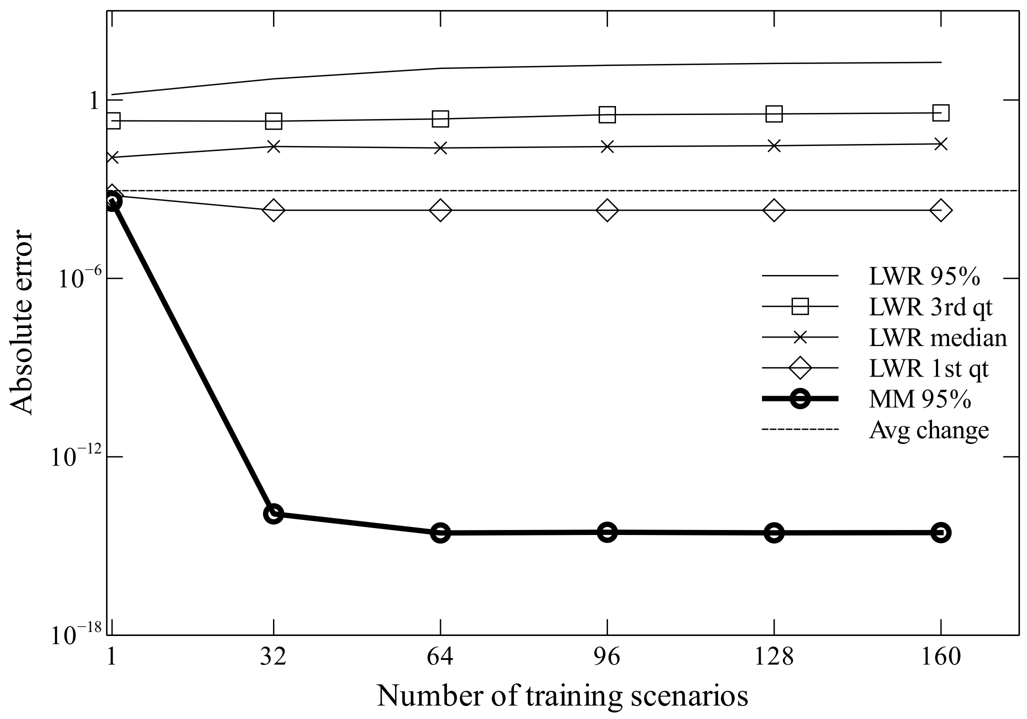


Figure 4.20: Comparison of prediction accuracy of our algorithm with uncentered LWR

4.7.2 Classifier accuracy

To demonstrate the benefit of using spatial relations to classify modes, we compare our combined FOIL/decision tree classifier to support vector machine (SVM) and K nearest neighbor (KNN) classifiers (*Hastie et al.*, 2002). We used the *MATLAB* (2010) built-in functions `svmtrain` and `svmclassify` with a quadratic kernel for training the SVM classifier. We set $K = 10$ for the KNN classifier. We found that using 10 neighbors is significantly better than using 1 neighbor, but more than 10 neighbors did not improve performance. All classifiers were trained and tested in the environment used in section 4.4, using the same set of training and test examples. The classification problem is to determine which of the modes listed in table 4.9 the ball exhibits in any example. To generate ground truth mode labels for each training and test example, we used the mode functions listed in table 4.9 to predict the outcome of each example. The mode function that gave the lowest prediction error was considered the true mode for that example. For both SVM and KNN classifiers, we centered the positions of all objects on the ball as we did in the LWR comparison, to increase the accuracy of those methods. We again averaged the data over 50 training batches consisting of 10 permutations of the configuration ordering, each repeated 5 times with different sets of random seeds. The results are shown in figure 4.21.

In the figure, the x axis marks the number of training scenarios seen, with each scenario consisting of 200 training examples as usual. The y axis as a log scale and marks the rate at which the classifier misclassified test examples, averaged over the 50 training batches. The plot shows that the FOIL classifier converges faster than the SVM and KNN classifiers, and also at a lower error rate. After 32 training scenarios, the error rate of the FOIL classifier is at approximately 4.1%, while that of the SVM classifier is at 17.7% and the KNN classifier is at 30.7%. The error rate of FOIL ends at around 3.5% after 160 training scenarios while that of SVM ends at 8.2% and KNN ends at 25.9%.

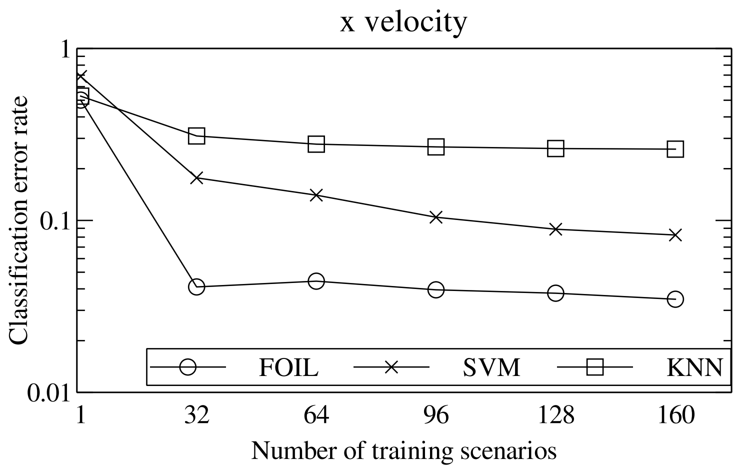


Figure 4.21: Comparison of accuracy for FOIL, SVM, and KNN classifiers.

We hypothesize that the reason for these differences in performance is that modes in spatial environments are determined primarily by the relationships between objects, not by their absolute positions in the coordinate system. The SVM classifier with its quadratic kernel can capture those relationships to some extent: the product of the coordinates of the ball and another object correlates to some degree to the distance between them, and hence whether they intersect, but the correlation is weak. The KNN classifier on the other hand only considers the positions of the objects in the coordinate system. Even though we centered the coordinate system on the ball and therefore correlate the positions of the other objects with their distance to the ball to some extent, the KNN classifier still does not correctly generalize over irrelevant characteristics such as the positions of objects far away from the ball that do not affect its behavior at all.

4.8 Model accuracy for multiple step predictions

In the previous experiments, we evaluated the accuracy of the learned models in making single step predictions. Many applications of action models such as planning and look-ahead search require making predictions multiple time steps into the future.

In a multi-step prediction, the model's prediction for time step i is based on the state at time step $i-1$, so prediction error accumulates over time. Specifically, if the model makes a large prediction error at time step i, all subsequent predictions will have at least that much error, even if individually they were perfectly accurate.

In this experiment we test the accuracy of the models learned in experiment 1 (ball-box-ramp domain without friction) in multi-step predictions. The model is trained in the same way as in experiment 1. When testing, the environment is initialized to one of the 40 initial configurations, and the model is used to predict the trajectory of the ball for 200 time steps. This is achieved as follows. At time step i, the system predicts the x and y velocities of the ball in time step $i+1$ using the learned models. It then predicts the x and y coordinates of the ball by adding the predicted velocities to the x and y coordinates at time i. The scene graph is then updated with the new position values, and the relational state is extracted for time $i+1$. This relational state is used for mode classification and role assignments in the prediction for the next time step. The model is tested on all 40 initial configurations, with 2 random seeds, for a total of 80 trials. All results were obtained from a single pair of x and y velocity models, since averaging multi-step predictions doesn't make sense. These were both the most accurate and most common models obtained in the 50 training batches from experiment 1.

Figure 4.22 and 4.23 plot the prediction errors for the x and y coordinates of the ball over time. The x axis indicates the time step into the prediction. The y axis indicates the absolute prediction error. Each line represents one series of 200 step predictions. The models predicted most of the steps accurately, but failed on a few types of interactions. Unfortunately, because of the cumulative nature of the predictions, an error at a single time step will ruin the accuracy of all future predictions, as the lines in the plots show. In total, 26 of the 80 trials contained one significant prediction error. We now discuss the types of prediction errors encountered.

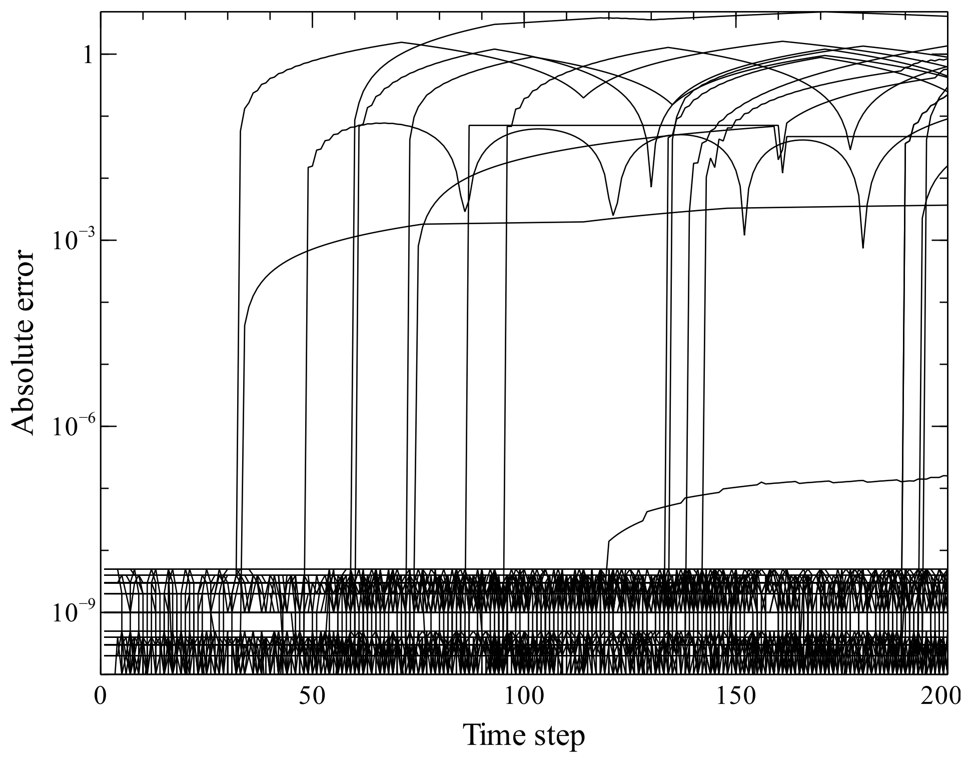


Figure 4.22: Extended prediction error for the x position of the ball

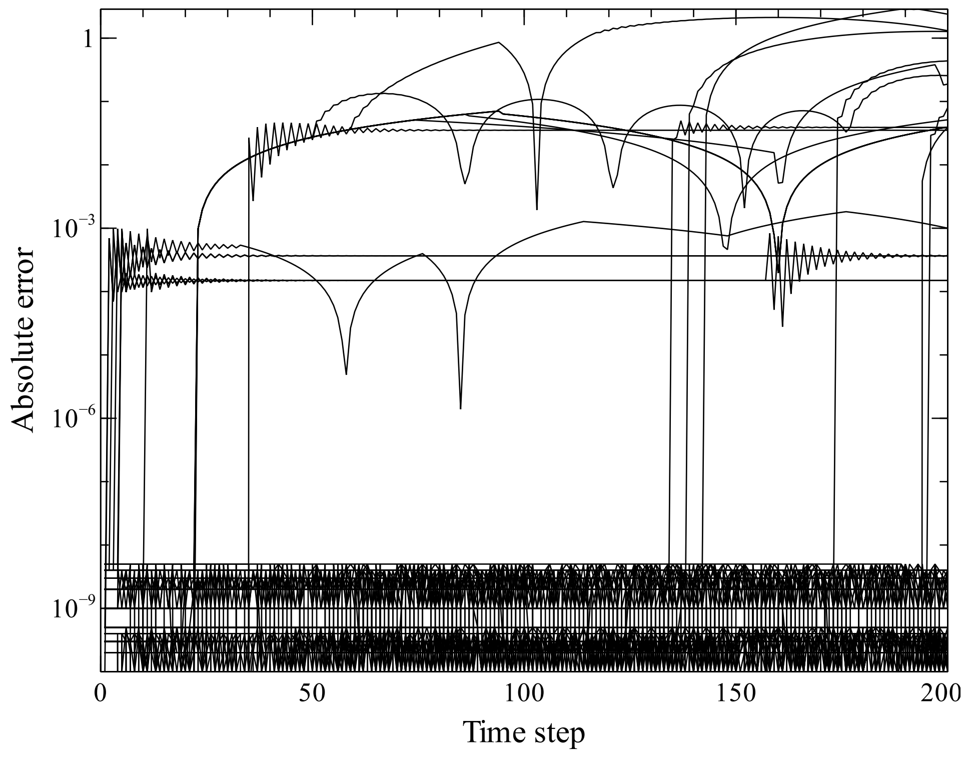


Figure 4.23: Extended prediction error for the y position of the ball

The first type of failure is due to an artifact of how the physics simulator handles collisions. Because the simulation is executed in discrete time steps, a collision between two objects is not detected until the time step when the two objects have overlapped. In the next time step, forces are exerted on the objects to push them apart in a process called collision resolution. The model therefore learns that whenever the ball intersects another object, it will exhibit the bouncing mode and its velocity will be reflected away from the object. In most cases this works correctly and the model predicts bounces accurately. However, in rare cases, collision resolution does not move objects sufficiently far apart in one time step, resulting in the objects still intersecting in the time step after the initial collision. The model has trouble with these cases because in the relational state space they resemble additional collisions. Therefore, when the ball is still intersecting an object in the second time step, the model predicts another bounce and another reflection in its velocity, this time toward the object being penetrated. This leads to the model predicting that the ball gets stuck in an infinite number of consecutive bounces. This type of error occurred when the ball bounced against a vertical surface, such as the side of the box or wall, or the slanted side of the ramp, but never when bouncing against the floor. Figure 4.24 illustrates this type of error. In the figure, each panel shows the actual (black) and predicted (gray) positions of the ball at a certain time step.

This type of prediction error occurred 22 times total in the experiment. 13 of those occurrences were when the ball bounced on a vertical surface, and the other 9 were when the ball bounced against the slanted side of the ramp. This is quite rare considering the ball bounced against vertical surfaces 205 times and the slanted side of the ramp 108 times total in the experiment. Because this type of error is arguably an artifact of the physics simulator and would not occur in the real world, we manually corrected them in the results shown in figures 4.22 and 4.23.

The second type of interaction that the model fails to predict correctly is the ball

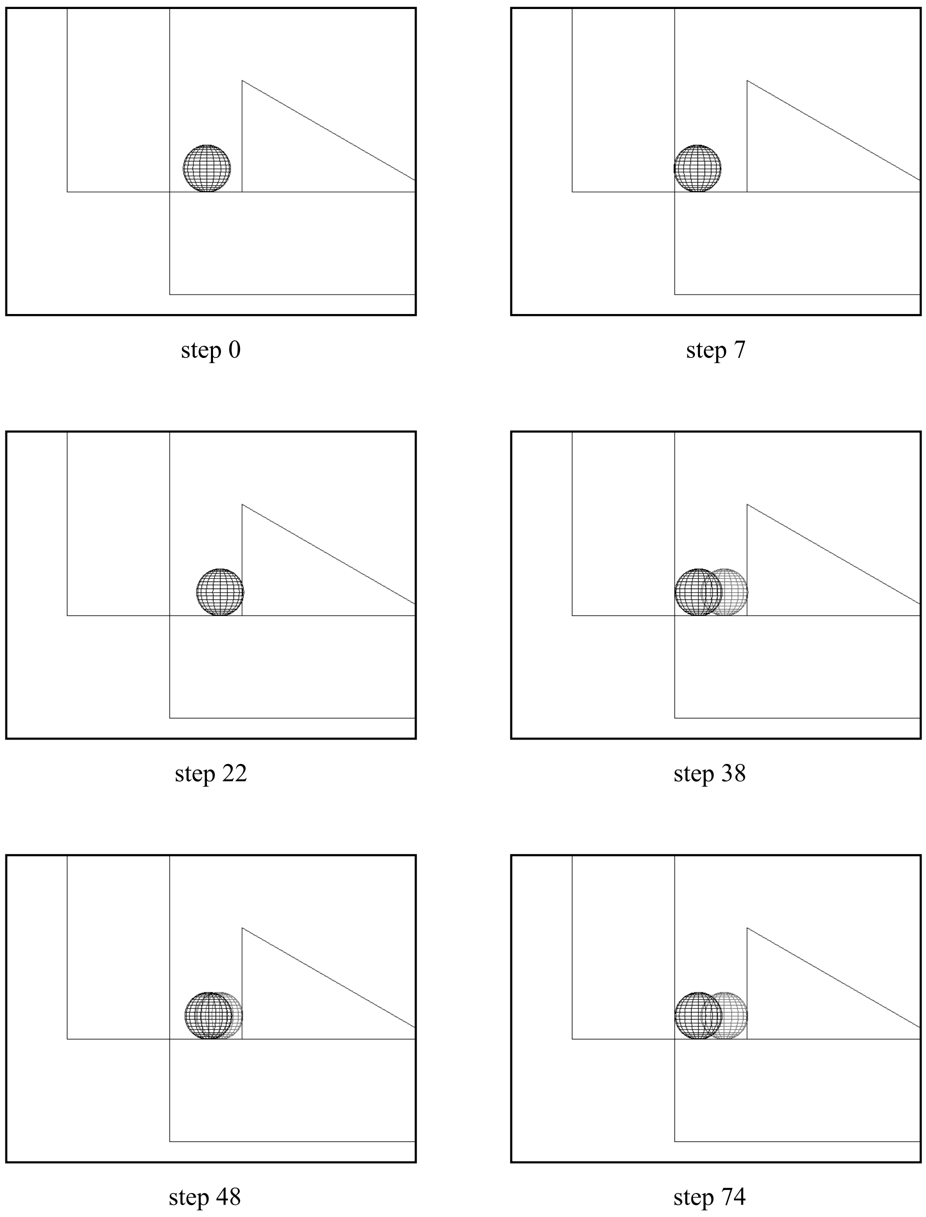


Figure 4.24: Actual (black) and predicted (gray) positions of the ball getting stuck in the wall due to imperfect collision resolution.

Error type	Num. Occurrences
Bad collision resolution, vertical	9 (out of 205 similar bounces)
Bad collision resolution, ramp slant	9 (out of 108 similar bounces)
Bounce against corner	4
Misclassified ramp bounce	5
Delayed falling	7

Table 4.18: Frequencies of prediction errors encountered in multi-step predictions.

bouncing against sharp corners. This is to be expected as the ball's exit velocity from this type of bounce is a non-linear function of the angle of incidence. In some cases this will result in a slight prediction error in the bounce, which is then propagated into future predictions, as in figure 4.25. In other cases, the error results in the ball getting stuck in the corner, much like the issue discussed previously, which results in more severe prediction errors, as shown in figure 4.26. This type of error occurred 4 times in the experiments.

The model experienced another type of failure when the ball bounced against the vertical side of the ramp while the mode classifier predicted it was in the mode of bouncing against the slanted side of the ramp. This results in a prediction of the ball penetrating deeper into the ramp's side and then getting stuck in an endless sequence of bounces. A final type of prediction error occurs as the ball slides off the top of the box: the model would predict that the ball begins falling one time step later than it should. This results in minor prediction inaccuracies but no qualitative error. The number of occurrences of these errors are listed in table 4.8.

4.9 Transfer to domains with different numbers of objects

One of the main advantages relational models have over propositional ones is that relational models can be applied to environments with different numbers of objects as the ones it was trained on. Since a change in the number of objects means a change in the number of dimensions in the continuous state vector, there is no way

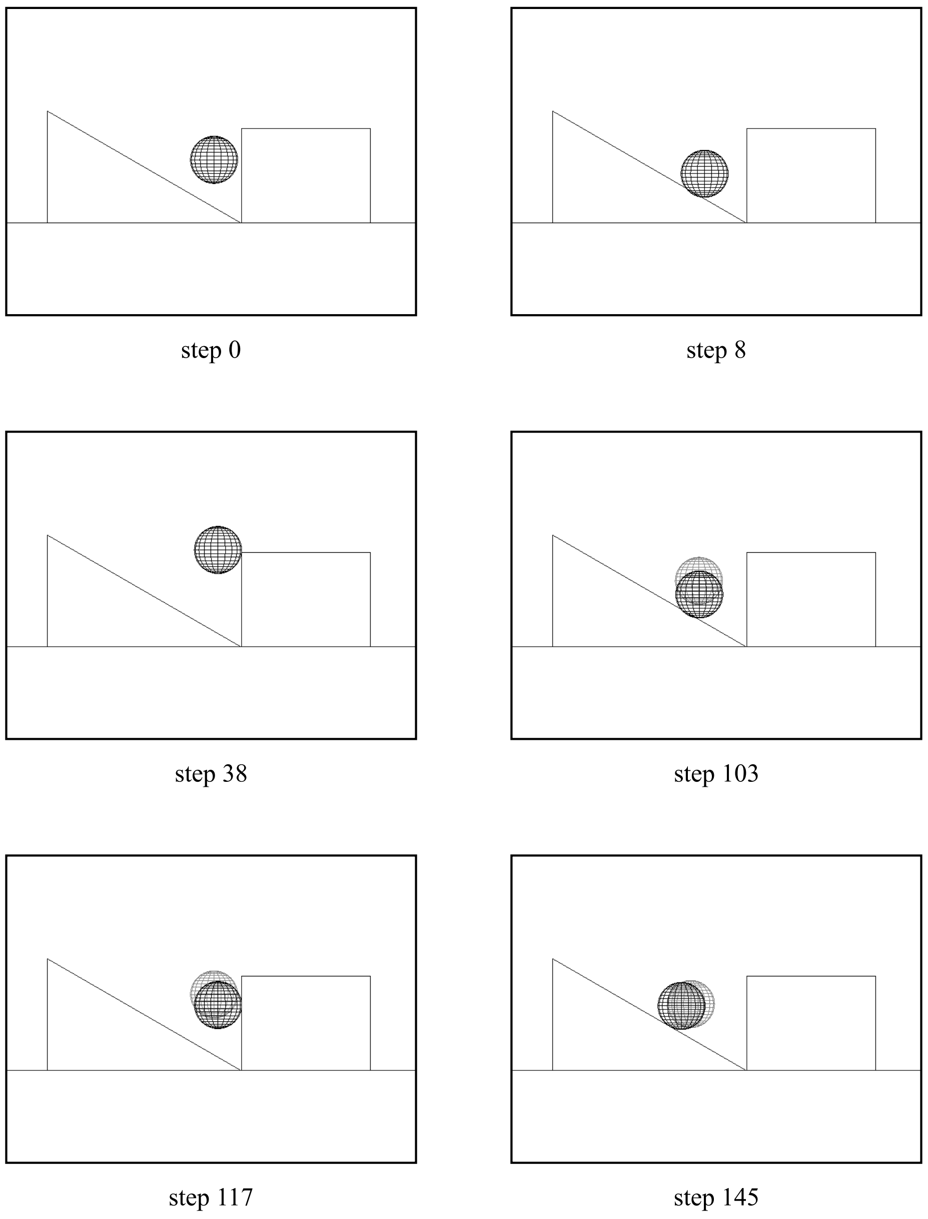


Figure 4.25: Actual and predicted positions of the ball bouncing against a corner, low prediction error.

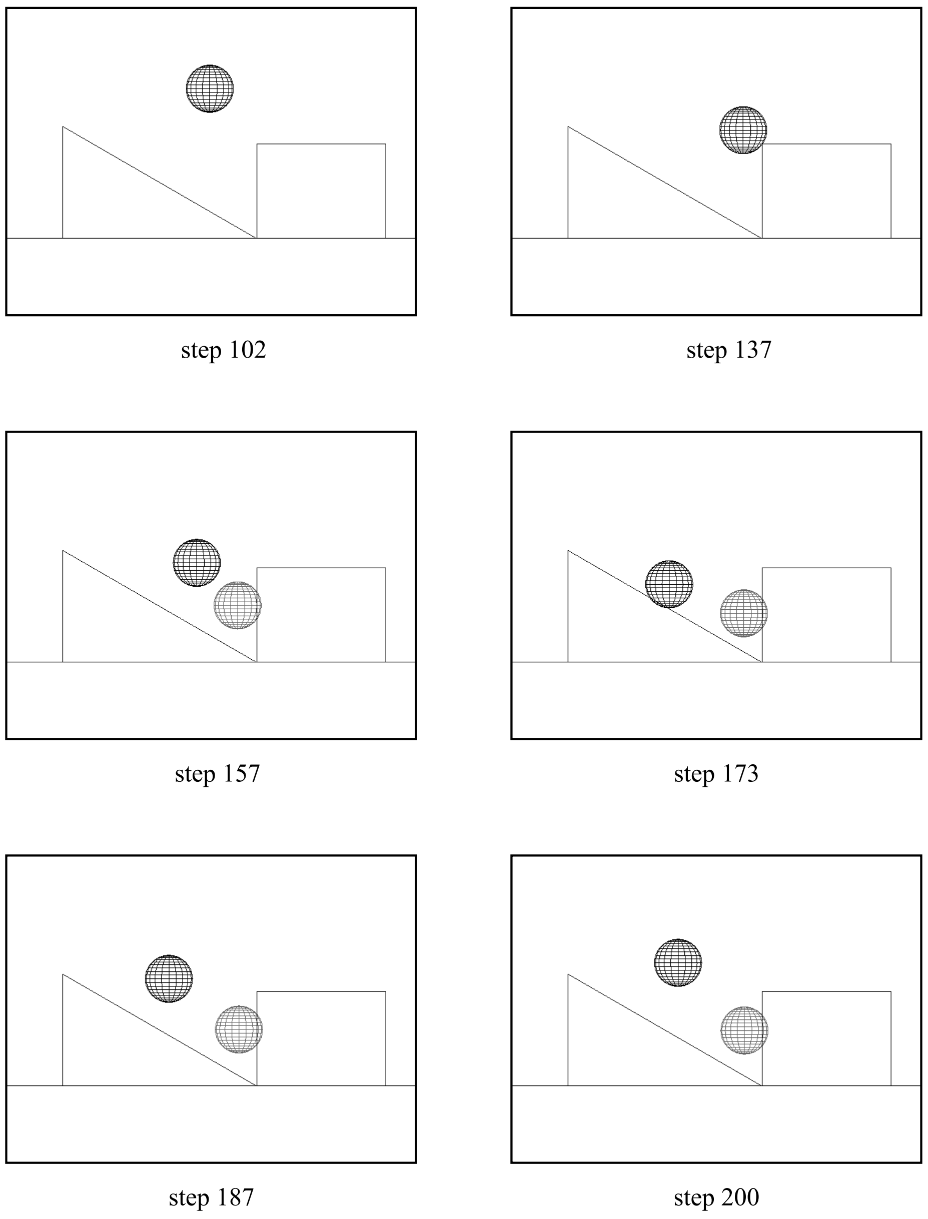


Figure 4.26: Actual and predicted positions of the ball bouncing against a corner, high prediction error.

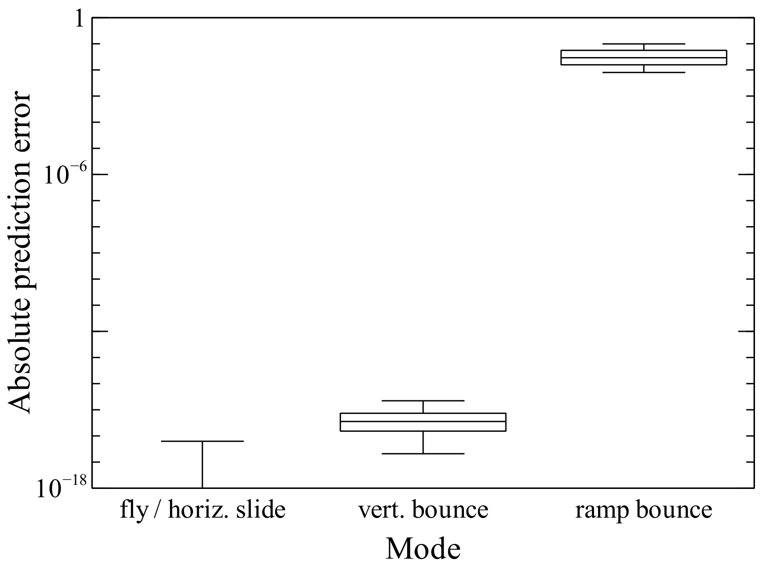


Figure 4.27: Prediction error for each mode of x velocity model trained in box-only environment.

a propositional model can handle this transfer without additional knowledge about how the dimensions in the training environment map to the dimensions in the test environment. On the other hand, our algorithm is able to handle such cases because the propositional linear models are generalized with relational role classifiers and mode preconditions.

In this experiment, we show how models transfer between environments with different numbers of objects. We again use the frictionless ball-box-ramp domain as our environment. We train models in environments with only the box or the ramp, and test its performance in an environment with both box and ramp. The training methodology is the same as in section 4.3, except the appropriate object is removed from the environment. Figures 4.27, 4.28, and 4.29 show the prediction errors of the models in the test environment (with both box and ramp). As before, the boxes indicate the first and third quartiles, the whiskers the 5 and 95 percentiles, and the line in the box the median of the data. The data was aggregated over 20 batches.

Figure 4.27 shows the prediction error of models trained in the box-only environment. As expected, the model learns the horizontal sliding/flying mode and the

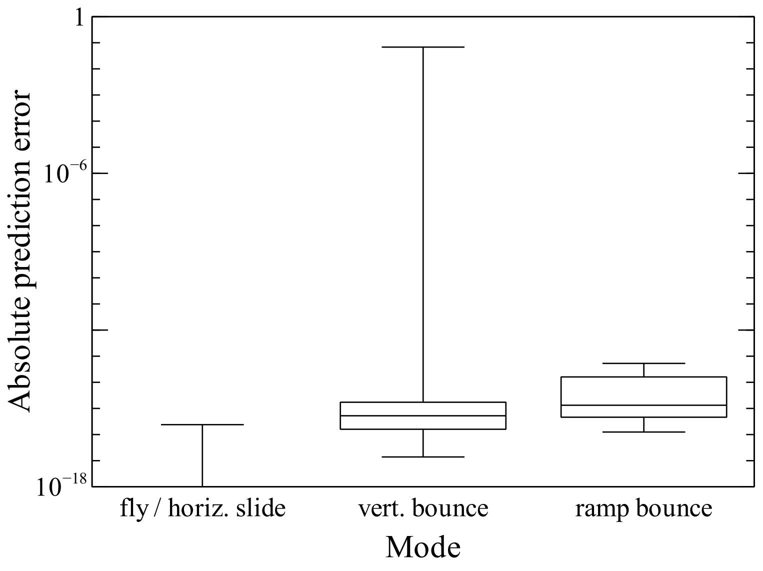


Figure 4.28: Prediction error for each mode of x velocity model trained in ramp-only environment.

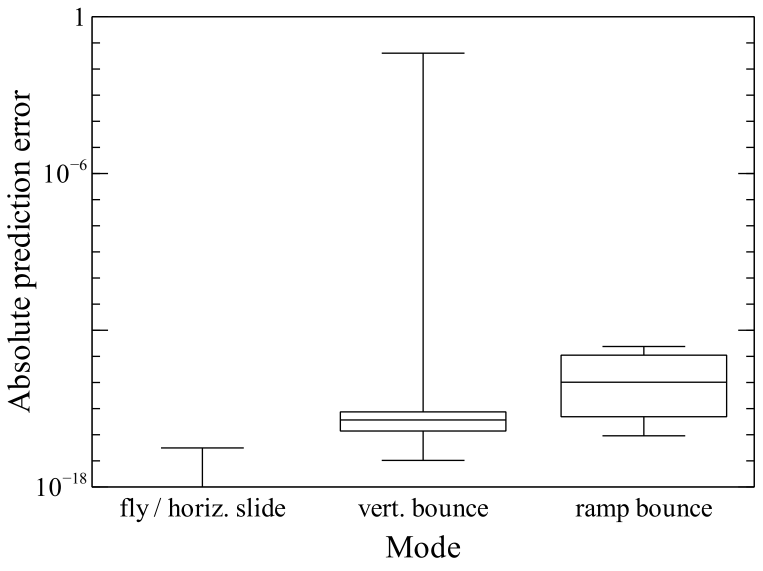


Figure 4.29: Prediction error for each mode of x velocity model trained in box-only, then ramp-only environments.

vertical bounce modes (modes 1 and 2 in table 4.2) correctly. It does not learn the ramp bouncing/sliding mode as it never encounters any examples of this. Therefore, the model's prediction error is low for the first two modes and high for the last mode. The point to note here is that the model is able to correctly predict the interactions that were present in its training data instead of failing catastrophically for all modes.

Figure 4.28 shows the prediction error of models trained in the ramp-only environment. The system learns all three modes listed in table 4.2 correctly. It was able to learn the vertical bouncing mode correctly because one side of the ramp has a vertical surface, as well as the walls. However, the prediction error for the vertical bouncing mode is higher than that of the model trained in the box-only environment because some of the learned mode classifiers misclassified vertical bounces against a box as ramp bounces. There were other cases where the system learned mode classifiers that were able to correctly predict vertical bounces against boxes, which demonstrates the generality of using first-order Horn clauses.

Figure 4.29 shows the prediction error of models first trained in the box-only environment, then in the ramp-only environment. Its accuracy is approximately equal to that of the models trained in just the ramp-only environment. Again, all modes were learned correctly. An unintuitive result is that the mode 2 prediction error is not lower than that when the model was just trained on the ramp-only environment. We hypothesize that since ramps and boxes never show up together in the training examples, FOIL does not add literals to clauses to distinguish between them, leading to classification errors.

CHAPTER V

Conclusion

In the previous chapters, we have developed an action modeling algorithm that exploits both the continuous and relational structure of spatial, object-based environments. We demonstrated the algorithm's performance in two environments, one of a truck pushing stacked boxes under the influence of friction, and a realistic physics simulation in which a ball bounces around a room with a box and a ramp. We showed that the algorithm outperforms Locally Weighted Regression in the ball-box-ramp domain. We also demonstrated how the algorithm performed when the environment signal was degraded with different amounts of Gaussian noise.

Our system exploits both continuous and relational regularities of spatial domains. Continuous regularities are embodied as modes of behavior in training data that describe qualitatively distinct interactions between objects as piecewise linear functions. Modes are discovered by clustering training examples in the continuous state space based on shared linear regularities. Relevant object roles in each mode are also explicitly identified in this process. The modes and roles discovered in the training data are then generalized into relational descriptions by learning classifiers that identify regularities in the relational state space shared by the modes and roles. General relational descriptions of when each mode manifests are learned using the FOIL algorithm and embodied in the mode classifier. The mode classifier predicts the correct mode to apply based on spatial relationships that are invariant to incidental encoding variations such as different coordinate systems, different object names, or different object scales. Relational role classifiers are learned that are able to assign the appropriate object to each role based on spatial relationships, decoupling modes from the specific objects in training examples. This allows mode functions to be applied in new contexts.

A strength of our algorithm is that it learns online and does not need a comprehensive training set before it can build models of individual modes. This is especially beneficial in domains where some modes are rarely encountered, such as the bouncing modes in the ball-box-ramp domain used in our experiments. The learner was able to achieve high predictive accuracy on common modes like flying and sliding early on, whereas uncommon modes like bouncing were identified later when a sufficient number of examples were collected. Overfitting due to committing to modes too early is corrected by unification.

The major contribution of this thesis is the development of an algorithm that learns continuous piecewise-linear action models with relational roles and preconditions. To our knowledge, our algorithm is the first that combines continuous propositional representations and learning techniques with relational representations and learning techniques for action modeling. These properties are important for learning general models with few training examples in spatial domains with multiple objects. Modeling changes of continuous properties is necessary for continuous environments, such as robotics tasks. It also avoids the ramification problem suffered by relational action modeling approaches. Having relational roles and preconditions allow learned models to predict the behavior of any object that satisfies relational constraints. This allows model learning to generalize faster when the same qualitative behaviors are repeated by different sets of objects or in different situations. More specific contributions are:

- Identifying the benefit of relational representations in continuous action models. Previous work in action modeling considered relational representations or propositional representations individually. Our system models the effects of actions in the continuous, propositional state representation most natural for continuous domains, but exploits first-order relational representations that compactly and naturally capture the higher-level structure of such domains. This increases the generality of the action models learned by our system.

- Novel application of FOIL to discover relational preconditions of modes. Representing preconditions as sets of first-order Horn clauses that test spatial relations aligns well with the regularities of object-based environments, where behaviors are conditioned on object interactions. We showed that purely continuous classifiers are not as accurate in distinguishing modes (section 4.7.2).

- A novel combination of relational and continuous classification. We showed that purely relational classifiers were not sufficiently discriminative in many circumstances, and developed a combined relational-continuous classification method.

- Identification of roles in propositional functions and learning descriptions of roles with FOIL. Roles generalize propositional linear functions so that they can apply to any objects that fit the role requirements. This allows a mode function learned for one set of objects to predict the behavior of another set of analogous objects, and to combine training examples from multiple sets of objects to train one model.

5.1 Future work

The work presented in this thesis considers action modeling as a stand-alone process. In real applications, action modeling would be integrated in an end-to-end agent architecture such as a robot. This would provide opportunities to extend the algorithm, such as by incorporating active learning techniques. Active learning is an important aspect of action modeling since the agent has direct control over which training examples it receives by choosing its actions, and can bias its actions to speed up learning. QLAP(*Mugan and Kuipers*, 2012) is an example of a system that performs active learning. Active learning can be incorporated into our system in many ways. For example, the agent can speed up mode discovery by actively seeking out training examples that don't belong to any existing modes. This can overcome the problem of uneven distributions of examples among modes, such as that encountered in the ball-box-ramp domain. The agent can also try to bring objects into specific relationships with each other to try to discover new modes of interaction, or use analogical reasoning to hypothesize the existence of modes for novel objects.

A complete agent will also use the learned model for planning. Sampling-based planning algorithms such as RRT (*LaValle and Kuffner*, 2001) are common in robotic domains, and these planners require fast roll-outs to keep planning time to a minimum. Even though prediction times for the models learned in our experiments were on the order of tens of milliseconds, they may grow significantly in more complex domains. Prediction in our model involves nontrivial operations including solving CSPs to find role assignments and extracting all spatial relations in the relational state. Many of these operations can be avoided by caching results between time steps. For example, we can assume that the same role assignments will persist across time steps until a different mode is encountered.

The learning portion of our algorithm presents more severe performance issues since many of the algorithms used are not incremental. Specifically, the FOIL learner used for learning role and mode classifiers takes on the order of tens of minutes to run in the worst cases in our experiments. These problematic cases arise when input into mode and role classifier learning components are overly complex. For example, if the environment signal is corrupted by a large amount of noise, our system may overfit the noise and learn a large number of modes. Since one binary classifier is learned for each pair of modes, this results in a quadratic increase in the number of times our system runs FOIL. Furthermore, since the modes are overfit and do not correspond to actual qualitative behaviors in the environment, there is usually not a clear set of clauses that can distinguish them, causing each run of FOIL to slow down further as it tries to build large clauses. An analogous situation can arise with role classifiers: an overfit mode function may have a large number of spurious roles, and FOIL must be run for each. These issues suggest that our system should have more safeguards against overfitting noisy data, although this usually leads to additional free parameters and accuracy trade-offs. Furthermore, the system should use an incremental inductive learner instead of FOIL (*Muggleton and Raedt*, 1994).

Another major shortcoming is that mode functions are restricted to linear functions. This restriction was made to avoid having to trade off between having fewer complex modes with having more simple modes. Higher order regression algorithms are more susceptible to overfitting than linear regression, and may cluster into single modes examples that belong in separate modes. On the other hand, when the environment exhibits truly non-linear modes, linear regression will either consider it to be noise, or overfit incidentally linear portions of the mode. The online learning context makes this balance even harder to establish, since the system doesn't have a representative global sample of the data that it can use to minimize overall error.

BIBLIOGRAPHY

BIBLIOGRAPHY

Aha, D. W., D. F. Kibler, and M. K. Albert (1991), Instance-based learning algorithms, *Machine Learning*, *6*, 37–66.

Akaike, H. (1974), A new look at the statistical model identification, *Automatic Control, IEEE Transactions on*, *19*(6), 716–723, doi:10.1109/TAC.1974.1100705.

Atkeson, C., A. Moore, and S. Schaal (1997a), Locally weighted learning, *AI Review*, *11*, 11–73.

Atkeson, C., A. Moore, and S. Schaal (1997b), Locally weighted learning for control, *AI Review*, *11*, 75–113.

Carbonell, J. G., and Y. Gil (1996), Learning by experimentation: The operator refinement method, *Machine Learning: An Artificial Intelligence Approach*, *3*, 191—213.

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

Fischler, M. A., and R. C. Bolles (1981), Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography, *Commun. ACM*, *24*(6), 381–395, doi:10.1145/358669.358692.

Forbus, K. D. (1984), Qualitative process theory, *Artificial Intelligence*, *24*, 85–168.

Getoor, L., and B. Taskar (Eds.) (2007), *Introduction to Statistical Relational Learning*, The MIT Press.

Ghallab, M., A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins (1998), Pddl - the planning domain definition language, version 1.2, *Tech. Rep. CVC TR-98-003*, Yale Center for Computational Vision and Control.

Ginsberg, M. L., and D. E. Smith (1988), Reasoning about action i: A possible worlds approach, *Artificial intelligence*, *35*(2), 165–195.

Hastie, T., R. Tibshirani, and J. Friedman (2002), *The Elements of Statistical Learning*, Springer.

Hsu, C.-W., and C.-J. Lin (2002), A comparison of methods for multiclass support vector machines, *IEEE Transactions on Neural Networks*, *13*(2), 415–425.

Huffman, S. B., and J. E. Laird (1992), Using concrete, perceptually based representations to avoid the frame problem, in *AAAI Spring Symposium on Reasoning with Diagrammatic Representations*.

Kaelbling, L. P., H. M. Pasula, and L. S. Zettlemoyer (2007), Learning symbolic models of stochastic domains, *Journal of Artificial Intelligence Research*, *29*, 309–352.

Kuipers, B. (1994), *Qualitative Reasoning*, The MIT Press.

Lange, K. (2010), *Numerical analysis for statisticians 2nd edition*, Springer, New York, NY [u.a.].

LaValle, S. M., and J. J. Kuffner (2001), Randomized kinodynamic planning, *The International Journal of Robotics Research*, *20*(5), 378–400.

Lembcke, S. (2013), Chipmunk2d, http://chipmunk-physics.net, accessed September 2013.

Ly, D. L., and H. Lipson (2012), Learning symbolic representations of hybrid dynamical systems, *Journal of Machine Learning Research*, *13*, 35853618.

MATLAB (2010), *version 7.10.0 (R2010a)*, The MathWorks Inc., Natick, Massachusetts.

Mitchell, T. M. (1997), *Machine Learning*, McGraw-Hill.

Moore, A. W., and C. G. Atkeson (1993), Prioritized sweeping: Reinforcement learning with less data and less time, in *Machine Learning*, pp. 103–130.

Mugan, J., and B. Kuipers (2012), Autonomous learning of high-level states and actions in continuous environments, *IEEE Transactions on Autonomous Mental Development (TAMD)*, *4*(1), 70–86.

Muggleton, S., and L. D. Raedt (1994), Inductive logic programming: Theory and methods, *The Journal of Logic Programming*, *19*, 629–679.

Nguyen-tuong, D., and J. Peters (2008), Local gaussian process regression for real time online model learning and control, in *In Advances in Neural Information Processing Systems 22 (NIPS)*.

Nguyen-Tuong, D., and J. Peters (2011), Model learning for robot control: a survey, *Cognitive processing*, *12*(4), 319–340.

Potts, D. (2005), Incremental learning of linear model trees, in *Machine Learning*, pp. 5–48, Springer.

Quinlan, J. R. (1990), Learning logical definitions from relations, *Machine Learning*, *5*, 239–266.

Quinlan, R. J. (1992), Learning with continuous classes, in *5th Australian Joint Conference on Artificial Intelligence*, pp. 343–348, World Scientific, Singapore.

Rasmussen, C. E., and C. K. I. Williams (2005), *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*, The MIT Press.

Russell, S. J., and P. Norvig (2003), *Artificial Intelligence: A Modern Approach Second Edition*, Pearson Education.

Schmidt, M., and H. Lipson (2010), Symbolic regression of implicit equations, in *Genetic Programming Theory and Practice VII*, edited by R. Riolo, U.-M. O'Reilly, and T. McConaghy, Genetic and Evolutionary Computation, pp. 73–85, Springer US.

Sutton, R. S. (1991), Dyna, an integrated architecture for learning, planning, and reacting, *SIGART Bulletin*, *2*(4), 160–163.

Toussaint, M., and S. Vijayakumar (2005), Learning discontinuities with products-of-sigmoids for switching between local models, in *Proceedings of the 22nd international conference on Machine Learning*, pp. 904–911, ACM Press.

Vijayakumar, S., A. D'souza, and S. Schaal (2005), Incremental online learning in high dimensions, *Neural Computation*, *17*(12), 2602–2634, doi:10.1162/089976605774320557.

Wang, X. (1995), Learning by observation and practice: An incremental approach for planning operator acquisition, in *Proceedings of the 12th International Conference on Machine Learning*, pp. 549–557, Morgan Kaufmann.

Xu, J. Z., and J. E. Laird (2010), Instance-based online learning of deterministic relational action models, in *Proceedings of the 24th AAAI Conference on Artificial Intelligence*.

Xu, J. Z., and J. E. Laird (2011), Combining learned discrete and continuous action models, in *Proceedings of the 25th AAAI Conference on Artificial Intelligence*.