Learning Games from Videos Guided by Descriptive Complexity

Łukasz Kaiser
LIAFA, CNRS & Université Paris Diderot
Paris, France

Abstract

In recent years, several systems have been proposed that learn the rules of a simple card or board game solely from visual demonstration. These systems were constructed for specific games and rely on substantial background knowledge. We introduce a general system for learning board game rules from videos and demonstrate it on several well-known games. The presented algorithm requires only a few demonstrations and minimal background knowledge, and, having learned the rules, automatically derives position evaluation functions and can play the learned games competitively. Our main technique is based on descriptive complexity, i.e. the logical means necessary to define a set of interest. We compute formulas defining allowed moves and final positions in a game in different logics and select the most adequate ones. We show that this method is well-suited for board games and there is strong theoretical evidence that it will generalize to other problems.

Introduction

Systems able to learn from visual observations are of central importance in many fields, especially in autonomous robotics and interactive computer vision. While there is a great amount of work dealing with object recognition and visual scene interpretation, only a few systems with the capacity for learning higher-level concepts have been presented thus far (Needham et al. 2005; Santos, Colton, and Magee 2006; Antanas et al. 2009; Barbu, Narayanaswamy, and Siskind 2010; Hazarika and Bhowmick 2011). In all these cases, experiments are performed on simple games, either card or board games. This domain is chosen because the number of visual objects is manageably small while there is still a lot of complexity in the interactions. Moreover, games are a natural model of many real-world interaction scenarios, making the results significant in a broader context.

To learn higher-level concepts from videos, it is usual to first derive a sequence of higher-level symbolic data from the video stream and then to learn from this data. Many techniques from computer vision are applicable in the first step, and different board game-playing robots have been shown and their construction has been widely discussed (see e.g. (Bailey, Mercer, and Plaw 2004) for a presentation of the issues involved and (Barbu, Narayanaswamy, and Siskind 2010) for a robot that integrates learning). But the second step – learning from the derived symbolic data – has not been investigated widely. As the problem is an instance of inductive logic programming (ILP), in most cases the systems relied on the ILP program Progol (Muggleton 1995). This choice was made because a relational and logical representation is well-suited for visual scenes and Progol is a very successful ILP system, benefiting from decades of development and able to learn game rules even for Chess (Goodacre 1996) when the data is manually prepared.

While Progol is an outstanding program, learning from data derived from vision systems is different than from manually prepared sets, and the above-mentioned visual learning systems succeed in learning only simple games. Moreover, each of them needs to include a non-trivial set of background knowledge rules and to optimize several parameters to make the ILP step work. For example, as discussed in Section VI in (Barbu, Narayanaswamy, and Siskind 2010), the system learning Tic-Tac-Toe must be given background arithmetic knowledge, board representation, the concept of a line, frame axioms, player and opponent predicates, piece ownership, board access predicates, and spatial predicates (over a dozen hand-crafted rules in total) and the authors needed to change the built-in predicate `at/4` to get reasonable efficiency. This is not satisfactory, and the complexity only increases when moving to more interesting games.

To be able to learn games such as Connect4 or Gomoku from short demonstration videos and with minimal background knowledge, we go back and investigate the basic assumptions of inductive logic programming. It is standard in logic programming to use Horn clauses and possibly-recursive predicates to represent the state of the system (e.g. the board and the pieces), interesting patterns of the state (e.g. a line of the same color), and temporal change (the moves). While there are many advantages of this approach, we argue that a more nuanced one is better suited for learning from visual data. First of all, we use relational structures and not formulas to represent the state of the system. This turns out to be a good fit for the vision system and reduces the amount of background knowledge we need: it suffices to recognize row, column and diagonal relations and different piece types on the board. Secondly, instead of relying on just one logic (Horn clauses and recursive predicates), we study several logics: pure first-order logic (FO), existential and guarded FO, and the transitive closure logic. Learning in

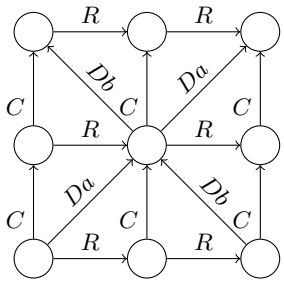

Figure 1: Relational representation of a 3×3 grid.

all of these logics and then selecting formulas yields better results and is more efficient than using any single one. These two fundamental changes allow us to demonstrate a system that – knowing only about rows, columns, diagonals and differentiating pieces – learns games like Connect4, Gomoku, Pawns, or Breakthrough, each one from a few intuitive video demonstrations, together around 2 minutes in length. We also integrate our algorithm with a general game playing (GGP) program (Kaiser and Stafiniak 2011) and thus the system can also play the learned games competitively. While our new learning techniques do not have the maturity of established ILP approaches, e.g. we do not implement hierarchically structured or probabilistic learning yet, they are arguably more adequate for learning from visual input.

State Representation and Visual Processing

We represent the state of the game in a fixed moment of time by a finite relational structure, which is the same as a labeled directed hypergraph. Formally, a relational structure $\mathfrak{A} = (A, R_1, \ldots, R_l)$ is composed of a universe A (denoted by the same letter in straight font) and a number of relations. We write r_i for the arity of the relation R_i, so $R_i \subseteq A^{r_i}$. The *signature* of $\mathfrak{A}$ is the set of symbols $\{R_1, \ldots, R_l\}$.

Game boards usually have a natural grid-like structure, and to represent them we use relational structures with four binary relations: R for the next-in-a-row relation, C for the next-in-a-column relation, and Da and Db for the two diagonals. The complete structure for the empty 3×3 grid, with 9 elements, is depicted in Figure 1. We use this structure to represent the starting position in Tic-Tac-Toe, and larger boards are represented in an analogous way. To mark pieces on the board, we use unary relations (predicates), e.g. a predicate Q for cross and P for circle. In all our experiments we represent game boards in exactly this way, but our learning algorithms work for arbitrary finite relational structures, thus also for more complex scenes and settings.

In the first step, our system reconstructs a sequence of relational structures, representing successive positions in the game, from each input video. We use off-the-shelf image processing methods in this step, and, as it is not the focus of this work, we describe them only briefly. At the start, we apply the Canny edge detector (Canny 1986) to the input video stream and use the Hough transform to detect lines in the standard way (Duda and Hart 1972). We try a few parameters for these operations and use the results to identify the size and position of the board in the frame and to detect the edges of pieces. We mark squares with no edges of pieces as empty, and for the rest we calculate the aggregate color within the edges of the potential piece, adjusted for the aggregate color of all pieces. Finally, based on this adjusted color, we assign the piece to one of the clusters (we used red, blue, yellow and black in the experiments) and mark the grid element with the appropriate predicate in the resulting structure. To determine when a move is made, we use a simple heuristic for hand detection based on the edges detected in the corners of the board.

The above methods for hand and board detection are not perfect and generate false board positions, especially during hand movement. To improve accuracy, we use the fact that only few predicates change in each move. We mark each board with more than two changed predicates as possibly-wrong. A sequence of frames with either a detected hand or a possibly-wrong board represents changes made on the board and is ignored, as we are interested only in the legal positions between such sequences. Among the frames between such sequences, we use majority voting to determine the one configuration of the board all these frames represent. The whole procedure was implemented using the OpenCV library for Canny edge detection and Hough transform and turned out to be sufficient to correctly reconstruct the plays of all games in our experiments.

Logic and Descriptive Complexity

Before we show how to derive interesting patterns from the sequences of structures reconstructed by the above procedure, we need to introduce some notions from descriptive complexity theory. This section presents the background necessary for this paper, refer to Chapter 3 of (Grädel et al. 2007) for a more complete introduction.

Recall that formulas of first-order logic over a relational signature $\{R_1, \ldots, R_l\}$ and with variables $x_1, x_2, \ldots$ ranging over elements of the structure have the form $\varphi :=$

$$R_i(x_1, \ldots, x_{r_i}) \mid x_i = x_j \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x_i\, \varphi \mid \forall x_i\, \varphi,$$

and their semantics, given an assignment of the variables x_i to elements e_i of the structure, is defined in the natural way, e.g. $\exists x_1 R(x_1, x_2)$ holds for an assignment $x_2 \to e_2$ in a structure $\mathfrak{A}$ if, and only if, there exists an element e_1 such that (e_1, e_2) is in the relation R in $\mathfrak{A}$. Notice that, for the grid structure presented in Figure 1, the formula $\neg\exists y\, C(y, x)$ holds exactly for elements from the bottom row. This formula, or the equivalent one $\forall y \neg C(y, x)$ in negation normal form, is a part of the winning condition in games where the goal of one of the players is to reach the bottom row.

First-order logic has several drawbacks from the computational point of view. First of all, it is not expressive enough to describe many relations that can easily be computed. This limitation stems from *locality* of first-order formulas. Intuitively, assume that a neighborhood of an element e in a structure consists of all elements connected to e by any of the relations, and a radius r neighborhood allows r-step connections. Then, a first-order formula can only detect whether certain patterns are present in the structure or not in neighborhoods of a fixed radius. This property, made precise in

the theorems of Gaifman (Gaifman 1982) and Hanf (Hanf 1965), implies that many patterns cannot be defined in FO.

Example 1. In our representation of the board (Figure 1) we allowed only the next-in-a-row relation R. Can we check that two elements are in the same row, but not necessarily next to each other? Indeed, for the 3×3 grid the formula $R(x,y) \vee \exists z(R(x,z) \wedge R(z,y))$ checks that x is left of y on the same row. But already a more complex formula is needed for a 4×4 grid, and the locality theorems imply that there is no FO formula expressing this property on all grids.

To remove this limitation of first-order logic, one extends FO by the *transitive closure operator* that allows to write formulas of the form $\mathrm{TC}\, x, y\, \varphi(x,y)$, which stands for the transitive and reflexive closure of the relation $\varphi(x,y)$. For example, $\mathrm{TC}\, x, y\, R(x,y)$ in our game board representation defines the relation "x is left of y in the same row" we considered above. In this work, we use a more precise operator that specifies exactly the number of steps to take in the transitive closure. Thus, we will write formulas of the form

$$\mathrm{TC}^m\, x, y\, \varphi(x,y),$$

for any first-order formula φ, and the semantics of such a formula is the set of all pairs (x,y) such that one can go from x to y in *exactly* m steps of the relation defined by $\varphi(x,y)$. For example, the formula $\mathrm{TC}^2\, x, y\, R(x,y)$ is equivalent to the first-order formula $\exists z(R(x,z) \wedge R(z,y))$.

Adding the transitive closure operator removes some limitations of FO, but how can we know what other problems remain? To answer this question, one can compare the expressive power of the resulting logic with computational complexity classes. For example, is every polynomial-time computable relation definable in the transitive closure logic, or some other extension of FO? Such questions are studied in *descriptive complexity theory*, and here we recall the most prominent known correspondences. The oldest result (Fagin 1974) shows that the class NP is captured by existential second-order logic. More practically, polynomial-time computations are captured by the least fixed-point logic (LFP) when a linear order relation is present (Immerman 1982; Vardi 1982). The requirement of a linear order can be weakened when a counting mechanism is added to the logic, and LFP with counting captures P on many classes of structures, such as grids, planar graphs (Grohe 1998) and all classes that exclude a fixed minor (Grohe 2010). Finally, while LFP is more expressive than the transitive closure logic (TC) we mentioned, TC captures all problems solvable in nondeterministic logarithmic space on ordered structures (Immerman 1987) and also on only locally (two-way) ordered graphs (Etessami and Immerman 2000).

The above results show what is the complexity of the patterns one can define in a logic, but they give little information about the complexity of *learning* formulas. Two most natural metrics for a formula are its size and its *quantifier rank*, i.e. the number of nested quantifiers inside a formula. For any logic $\mathcal{L}$, one can thus state the following problem: Given two finite relational structures, find an $\mathcal{L}$-formula of minimal quantifier rank (or size) distinguishing these structures. Unluckily, already for first-order logic the above problem is hard, namely PSPACE-complete (Pezzoli 1998). But there is a natural restriction of first-order logic, the k-variable fragment FO^k, for which this problem becomes solvable in polynomial time (Grohe 1999).

The k-variable fragment of FO consists of all formulas that use only the variables $x_1, \ldots, x_k$, both as free ones and under quantifiers. Note that variables under quantifiers can be renamed, e.g. the formula $\exists x_2(R(x_1,x_2) \wedge \exists x_1 C(x_2,x_1))$ belongs to the 2-variable fragment. When restricted to the k-variable fragment, one must ask whether a formula distinguishing two given structures exists at all in this fragment. Maybe the pattern of interest requires more than k variables? Luckily, for many classes of structures, a constant number of variables is sufficient. These include planar graphs, classes of graphs excluding a minor, and several other classes, see (Pikhurko and Verbitsky 2010) for a survey. The structures we use to represent game boards are planar, and therefore also fall into this category. Let us stress that the restriction to a low number of variables is one key reason why our learning algorithms are efficient, and the above results imply that this will still be the case for more complex structures. Thus, we conjecture that the methods we present below will generalize from board games to various other situations.

Distinguishing Relational Structures

In this section we present our main learning procedure that, given two sets of structures, the positive and the negative ones, returns a formula φ that holds on all positive structures and on none of the negative ones. As motivated above, the returned formula belongs to the k-variable fragment of first-order logic with the TC^m operator. The formula uses the minimal number of variables k, has minimal quantifier rank among k-variable formulas distinguishing the two sets of structures, belongs to the guarded fragment if possible, and is existential if possible (we will explain these notions and illustrate why they help in learning games later). The procedure runs in polynomial time if each input structure belongs to one of the classes mentioned above, in particular always when the input structures are planar. An important part of the procedure is the computation of $\mathcal{L}$*-types* of tuples of elements from the structures.

Definition 2. The $\mathcal{L}$-type of a tuple $\overline{a}$ in a structure $\mathfrak{A}$ is the subset of formulas of $\mathcal{L}$, with as many free variables as $|\overline{a}|$, that are satisfied by $\overline{a}$ in $\mathfrak{A}$, i.e.

$$\mathcal{L}\text{-type}(\mathfrak{A}, \overline{a}) = \big\{\varphi(\overline{x}) \in \mathcal{L} \;\mid\; |\overline{x}| = |\overline{a}| \text{ and } \mathfrak{A} \models \varphi(\overline{a})\big\}.$$

The set described above is most often infinite for trivial reasons, e.g. it might contain formulas $P(x), P(x) \wedge P(x), P(x) \wedge P(x) \wedge P(x)$, and so on – something that could be described just by $P(x)$. Since in many cases there exists one formula describing this set, we will often abuse the terminology and say that the $\mathcal{L}$-type of $\overline{a}$ in $\mathfrak{A}$ is a single formula $\tau \in \mathcal{L}$, denoted $\tau = \mathrm{tp}^{\mathcal{L}}(\mathfrak{A}, \overline{a})$, such that:

$$\mathfrak{A} \models \tau(\overline{a}) \text{ and for all } \varphi \in \mathcal{L}\text{-type}(\mathfrak{A}, \overline{a}) \text{ holds } \tau(\overline{x}) \Rightarrow \varphi(\overline{x}).$$

Note that, in principle, such a formula τ might not exist in the logic $\mathcal{L}$. But it does exist for fragments of FO that we consider here, e.g. for bounded quantifier rank, bounded number of variables, and for the guarded fragment.

Computing first-order types

For a fixed number of variables k and a bound n on the quantifier rank, let us denote by $\mathcal{L}^{n,k}$ the set of all first-order formulas using only the variables $x_1, \ldots, x_k$, i.e. from the k-variable fragment, and with quantifier rank at most n. Given a structure $\mathfrak{A}$ and a tuple $\overline{a}$ of length k, we will compute the $\mathcal{L}^{n,k}$-type of $\overline{a}$ inductively and denote the result $\mathrm{tp}^{n,k}(\mathfrak{A}, \overline{a})$.

For $n = 0$, the formula $\mathrm{tp}^{n,k}(\mathfrak{A}, \overline{a})$ is simply a conjunction of all literals satisfied by $\overline{a}$ in $\mathfrak{A}$, which we compute exhaustively. These are often long formulas with few positive atoms, e.g. in the structure in Figure 1 the $0, 2$-type of the pair of the bottom-left element and the central element is

$$Da(x_1, x_2) \wedge \neg Da(x_1, x_1) \wedge \neg Da(x_2, x_1) \wedge \neg Da(x_2, x_2) \\ \wedge \bigwedge_{v,w \in \{x_1, x_2\}} \neg Db(v, w) \wedge \neg C(v, w) \wedge \neg R(v, w).$$

For $n > 0$, the type $\mathrm{tp}^{n,k}(\mathfrak{A}, \overline{a})$ can be computed inductively, as it is given by the following formula:

$$\mathrm{tp}^{n-1,k}(\mathfrak{A}, \overline{a}) \wedge \bigwedge_{i < |\overline{a}|} \left(\forall x_i \left(\bigvee_{b \in \mathfrak{A}} \mathrm{tp}^{n-1,k}(\mathfrak{A}, \overline{a}[a_i \leftarrow b]) \right) \right. \\ \left. \wedge \bigwedge_{b \in \mathfrak{A}} \exists x_i \left(\mathrm{tp}^{n-1,k}(\mathfrak{A}, \overline{a}[a_i \leftarrow b]) \right) \right),$$

where $\overline{a}[a_i \leftarrow b]$ denotes the tuple $\overline{a}$ with the i-th element replaced by b. We omit the proof of correctness of this formula here, as it is very similar to the standard proof for FO.

Guarded types for sparse structures

In our procedure, we need to compute the types of all tuples in the structure. Even for 2-variable tuples on an 8×8 grid, this means computing the types for $64^2 = 4096$ tuples, which is slow, and for triples on a 19×19 grid it is not practical any more (though one could do it in parallel on multiple machines). But most of these tuples will be of no use for distinguishing structures because, aside from unary relations, they all have the same type: not connected by any relation. The structures we use to represent boards are *sparse* and thus, in almost all practical cases, at least one distinguishing tuple will be connected by some binary relations in the structure. This property has also been studied and exploited in logic – the fragment of first-order logic that requires tuples to be connected is called the *guarded fragment* and it is the main reason why modal and description logics enjoy good algorithmic properties (Grädel 1999).

Definition 3. The guarded fragment of FO is defined inductively as a syntactic subset given by the following grammar.

$$\varphi \ ::= R_i(x_1, \ldots, x_{r_i}) \mid x = x \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \\ \exists \overline{y} \big(R_i(\overline{x}, \overline{y}) \wedge \varphi(\overline{x}, \overline{y}) \big) \mid \forall \overline{y} \big(\neg R_i(\overline{x}, \overline{y}) \vee \varphi(\overline{x}, \overline{y}) \big),$$

where $\varphi(\overline{x}, \overline{y})$ means that *all* free variables of φ must be included in the set $\{\overline{x}\} \cup \{\overline{y}\}$.

Example 4. Formulas of modal logic translate to guarded first-order logic formulas with two variables. For example, a formula with one free variable x_1 expressing "every C-successor of x_1 has an R-successor in which P holds" can be written in the guarded fragment with two variables as:

$$\forall x_2 \big(C(x_1, x_2) \rightarrow \exists x_1 (R(x_2, x_1) \wedge P(x_1)) \big).$$

Again, for a fixed number of variables k and a bound n on quantifier rank, we denote by $\mathcal{G}^{n,k}$ the set of all guarded first-order formulas using only the variables $x_1, \ldots, x_k$, i.e. from the k-variable fragment, and with quantifier rank at most n. Given a structure $\mathfrak{A}$ and the tuple $\overline{a}$ of length k, we will compute the $\mathcal{G}^{n,k}$-type of $\overline{a}$ inductively and denote the result $\mathrm{tp}_G^{n,k}(\mathfrak{A}, \overline{a})$.

For $n = 0$ we have $\mathrm{tp}_G^{0,k}(\mathfrak{A}, \overline{a}) = \mathrm{tp}^{0,k}(\mathfrak{A}, \overline{a})$ as there is no difference between full and guarded logic.

For $n > 0$, the construction is different: Instead of quantifying over one variable, we find sets x of variables which can be used in a guard, and quantify over those variables. To this end, we first need to compute all *guarded substitutions* of the tuple $\overline{a}$. We say that $\overline{b}$ is a guarded substitution of $\overline{a}$ if $|\overline{b}| = |\overline{a}|$ and the following holds: There exists a subset $\{b_1, \ldots, b_k\}$ of $\overline{b}$ such that $(b_1, \ldots, b_k) \in R_i$ for some R_i, at least one $b_i \in \overline{a}$, and on all positions $j < |b|$ either $b[j] = a[j]$ or $b[j] = b_i$ for some i.

Let now S be the set of all guarded substitutions of $\overline{a}$ and V the set of all proper subsets of variables $x_1, \ldots, x_{|a|}$. For each non-empty set $\mathsf{x} \in V$ let G_x denote proper guards for x, i.e. formulas $R(\overline{x}, \overline{y})$ such that $\{\overline{x}\} = \mathsf{x}$ and $\overline{y}$ is not empty. For each such $g \in G_\mathsf{x}$ let us denote by S_g the subset of S for which the guard g holds, $S_g = \{\overline{b} \in S \mid \mathfrak{A} \models g(\overline{b})\}$. We define the next guarded type for x and $g \in G_\mathsf{x}$ as

$$\tau_{\mathsf{x},g} \ = \ \forall \mathsf{x} \left(g \rightarrow \bigvee_{\overline{b} \in S_g} \mathrm{tp}_G^{n-1,k}(\mathfrak{A}, \overline{b}) \right) \\ \wedge \bigwedge_{\overline{b} \in S_g} \exists \mathsf{x} \left(g \wedge \mathrm{tp}_G^{n-1,k}(\mathfrak{A}, \overline{b}) \right).$$

Finally, the guarded type $\mathrm{tp}_G^{n,k}(\mathfrak{A}, \overline{a})$ is given by

$$\mathrm{tp}_G^{n-1,k}(\mathfrak{A}, \overline{a}) \ \wedge \bigwedge_{\mathsf{x} \in V} \bigwedge_{g \in G_\mathsf{x}} \tau_{\mathsf{x},g}.$$

Again, we omit the proof that $\mathrm{tp}_G^{n,k}$ is indeed the $\mathcal{G}^{n,k}$-type, as it follows the above construction in a standard way.

An even more restricted logic than the n, k guarded fragment is the n, k *existential* guarded fragment, denoted $\mathcal{EG}^{n,k}$ and defined as all formulas from $\mathcal{G}^{n,k}$ in negation normal form in which no universal quantifier occurs. The above formulas allow to compute existential guarded types as well, only the whole universally quantified part must be removed.

Distinguishing positive and negative structures

Let P be a set of positive structures to be distinguished from the set N of negative ones. For a fixed logic $\mathcal{L}$, variable number k and quantifier rank n, the $\mathcal{L}^{n,k}$-distinguishing procedure proceeds as follows. First, it computes the set $\mathcal{N}$ of $\mathcal{L}^{n,k}$-types of all tuples in all structures in N. Then, for every structure $\mathfrak{A} \in P$, it finds an $\mathcal{L}^{n,k}$-type $\tau_\mathfrak{A}$ of some tuple $\overline{a}$

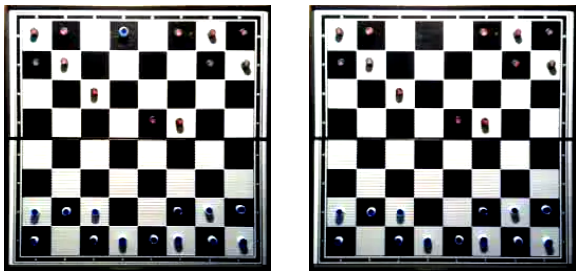

Figure 2: A position winning for white and one not winning.

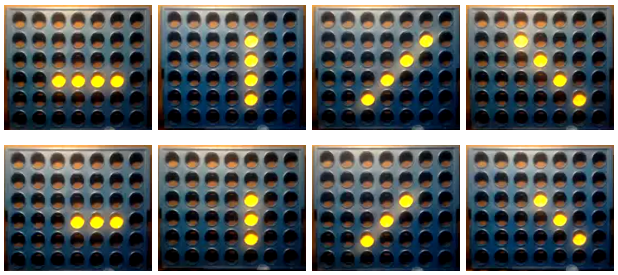

Figure 3: 4 positions winning for yellow and 4 not winning.

in $\mathfrak{A}$ such that $\tau_{\mathfrak{A}} \notin \mathcal{N}$. The formula $\varphi = \bigvee_{\mathfrak{A} \in P} \tau_{\mathfrak{A}}$ holds in each structure in P, because of the corresponding disjunct, and in no structure from N, because then some $\tau_{\mathfrak{A}}$ would belong to $\mathcal{N}$. Therefore φ distinguishes P from N.

The $\mathcal{L}^{n,k}$-distinguishing procedure described above is used iteratively, starting from the smallest k, for each k from the smallest n, and with fixed n and k starting from the weakest logic: first the existential guarded fragment, then the full guarded fragment, and only finally the full k-variable fragment with quantifier rank n. Additionally, for each k, after the atomic $0, k$-types τ have been computed we check whether, for some m, the formula $\text{TC}^m\, x_1, x_2\, \tau(x_1, x_2)$ distinguishes P from N. This allows to detect basic transitive relations efficiently. The complete `Distinguish` procedure is summarized below.

Procedure `Distinguish(`P, N`)`

$k \leftarrow 1$
while *FO*^{0,k} *does not distinguish* P *from* N **do**
  Try to distinguish P from N with TC formulas
  for $n = 0, \ldots, k+1$ **do**
    Try to $\mathcal{EG}^{n,k}$-distinguish P from N
    Try to $\mathcal{G}^{n,k}$-distinguish P from N
    Try to FO^{n,k}-distinguish P from N
  end
  $k \leftarrow k+1$
end

The above procedure finds formulas distinguishing P from N with minimal number of variables and minimal quantifier rank, but since the $\mathcal{L}^{n,k}$-distinguishing procedure relies on types, the returned formulas are normally very long and hard to read. We correct this by changing the $\mathcal{L}^{n,k}$-distinguishing procedure in the following way. Instead of returning $\varphi = \bigvee_{\mathfrak{A} \in P} \tau_{\mathfrak{A}}$, we will return $\varphi^{\min} = \bigvee_{\mathfrak{A} \in P} \tau^{\min}_{\mathfrak{A}}$, where $\tau^{\min}_{\mathfrak{A}}$ is computed as follows. From all types $\tau^1_{\mathfrak{A}}, \ldots, \tau^l_{\mathfrak{A}}$ which hold for some tuple $\overline{a}$ in $\mathfrak{A}$ but are not in $\mathcal{N}$ (computed previously as well), we greedily remove all literals that are not necessary to distinguish $\mathfrak{A}$ from N. The formula $\tau^{\min}_{\mathfrak{A}}$ is then the shortest of the remaining formulas. In our experiments, it usually turned out to be an easily readable one as well.

Learning Winning Conditions in Games

The procedure `Distinguish` described above is already sufficient to learn the winning conditions for both players in the games we experimented with. Consider for example the game of Breakthrough in which the goal of the white player is to get to the last row. An example of a winning position is depicted on the left in Figure 2, while the same position without the winning piece is on the right. Let $\mathfrak{A}^+$ be the 8×8-grid structure analogous to the one in Figure 1 but representing the board on the left in Figure 2, and let $\mathfrak{A}^-$ represent the board on the right, with white pieces marked by W and the black ones by B. Running `Distinguish`$(\{\mathfrak{A}^+\}, \{\mathfrak{A}^-\})$ results in the formula:

$$\exists x_1 (W(x_1) \wedge \forall x_0 \neg C(x_1, x_0)),$$

which expresses that there is a white piece in the last row.

In Figure 3, we give another example of 4 positive structures P and 4 negative structures N, this time representing configurations of a 7×6 grid corresponding to winning and not winning positions in Connect4, with Q for yellow. This time, the procedure `Distinguish`(P, N) returns

$$\exists x_0, x_1 \Big(\text{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge C(x_0, x_1)\right) \\ \vee \text{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge Da(x_0, x_1)\right) \\ \vee \text{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge Db(x_0, x_1)\right) \\ \vee \text{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge R(x_0, x_1)\right) \Big)$$

as it finds the transitive closures of Boolean combinations of literals distinguishing P from N for $k = 2$ variables.

Learning Legal Moves

Having learned the winning conditions, we still face the problem of determining which moves are legal and which are not. Since from each video we derive a sequence of structures, and since the underlying grid does not change, we can simply take the symmetric difference of the labels of two successive structures and get a *prototype* of a move: the two sub-structures containing only the elements that changed labels. For example, for Connect4 there would be only 2 prototypes of moves: changing a blank field to a red one, and changing it to a yellow one.

We derive the prototypes of moves from all available sequences, and thus the derived prototypes always cover all

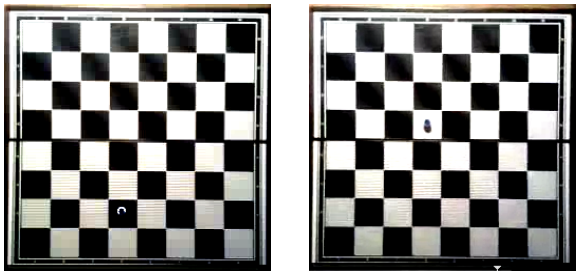

Figure 4: An illegal pawn move.

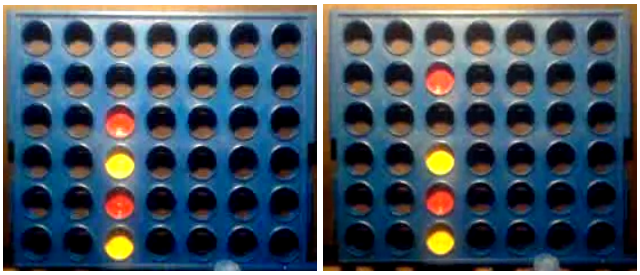

Figure 5: A legal and an illegal move in Connect4.

presented moves. In some cases, e.g. in Gomoku, the prototypes are already exactly the desired moves. But in most cases the prototypes are too general, as not every imaginable move is legal. For example, in Connect4 it is not possible to change a blank field to a yellow one or to a red one if there is still a blank field below it. To demonstrate such situations, we also record videos of *illegal* moves, such as the move of a pawn presented in Figure 4. Note that a move always consists of two structures.

For every generated move prototype, we gather all pairs of structures in which this move was applied legally, and also all pairs in which it was demonstrated as illegal. From each of these pairs, we take the first structure (the one before the move was applied) and add to it new predicates, marking the elements of the prototype, i.e. the fields on which the predicates change. After such marking, we again have a set of positive structures (the marked first ones from the legal moves) and a set of negative ones. This allows us to again use the `Distinguish` procedure to derive the precondition of a legal move. For example, consider Figure 5 in which we present an example of an outcome of a legal and of an illegal move. The field on which the upper red token is placed is the one that changes, so it gets marked by e_1. Let us denote the structures representing these two marked positions – the legal one, depicted on the left in Figure 5, and the illegal one, depicted on the right – by $\mathfrak{A}^+$ and $\mathfrak{A}^-$, respectively. Running `Distinguish`$(\{\mathfrak{A}^+\},\{\mathfrak{A}^-\})$ results in the following formula:

$$\exists x_1\big(Q(x_1) \wedge \exists x_0(C(x_1,x_0) \wedge x_0 = e_1)\big),$$

where e_1 is a constant marking the single element of the move prototype. This formula expresses that there must be a yellow element below the changed one.

Summary of Experimental Results

To learn a complete game, we use four kinds of videos. For the winning conditions, we use videos that present plays ending in positions won by the first player and some with plays ending in positions won by the second player. Additionally, it is convenient to allow videos that depict unfinished or tied plays, and, to distinguish legal and illegal moves, we may need illegal move videos. There are therefore 4 possible kinds of videos. The number of videos of each kind that we used to learn the correct rules of each of the example games is given in Table 1.

We ran the tests on a laptop with 4GB RAM and a 2.13GHz Intel L9600 processor. Our program used only a single processor core and the times needed for video processing and game learning are given in Table 2. The computation of move constraints in Pawn Whopping used around 6GB of memory and needed to swap it to disk, thus the long running time. All the videos, text transcripts of the sequences of structures, generated games, as well as the source code are available at `toss.sf.net` and will be included in the next release 0.8 of (Toss).

As far as the logics are concerned, the TC^m operator turned out to be useful in several games: in Connect4, in Gomoku and in Tic-Tac-Toe. All other derived formulas belong to the guarded fragment. In some games, e.g. in Connect4, formulas for the winning condition were quantifier-free and used the TC^m operator, while preconditions of some moves were expressed in the guarded fragment. This confirms that it is useful to consider different logics and to choose the suitable one algorithmically. Let us also emphasize that exploiting various logics is essential for efficiency: trying to simply find formulas in the k-variable fragment not only takes orders of magnitude more time (we stopped the tests after several hours), but also, even for simple games like Tic-Tac-Toe, it results in irregular formulas which are hard to read and to understand.

The video recognition procedure was implemented in C++ and the game learning algorithm in OCaml, and both were integrated with (Toss), an open-source general game playing program which also includes various logic functions. Due to this integration, the learning procedure outputs games in a format compatible with Toss. Therefore one can directly play all the learned games from Table 1, and the playing strength turned out to be exactly the same as for manually written definitions, which were tested and discussed in (Kaiser and Stafiniak 2011).

	1 Wins	2 Wins	Not Won	Illegal
Breakthrough	1	1	3	0
Connect4	4	4	13	4
Gomoku	4	4	9	0
Pawn Whopping	1	1	4	6
Tic-Tac-Toe	4	4	17	0

Table 1: Number of videos needed for each game.

	Video Processing	Game Learning
Breakthrough	48 s	120 s
Connect4	68 s	98 s
Gomoku	41 s	16 s
Pawn Whopping	74 s	906 s
Tic-Tac-Toe	28 s	8 s

Table 2: Running times of the procedures (in seconds).

Outlook

We presented a system that learns games such as Connect4, Gomoku, Pawns, or Breakthrough from short demonstration videos and with minimal background knowledge. Representing states as relational structures turned out to be very adequate and reduced the amount of background knowledge needed in the system. But our main contribution was to review the basic assumptions of inductive logic programming, especially the question which logic is best suited for representing and learning patterns efficiently. Guided by results from descriptive complexity theory, we decided for the k-variable fragment of first-order logic, reducing it to the guarded fragment whenever possible to improve efficiency and sometimes adding the transitive closure operator to increase expressive power. This combination allowed to generate very short and intuitive formulas in the experiments we performed, and there is strong theoretical evidence that it will generalize to other problems. Some of those problems might require hierarchical, structured learning or a form of probabilistic formulas, and in the future we intend to consider such extensions. But already the presented technique significantly improves the state of the art in learning from visual input.

Acknowledgment. We would like to thank Sasha Rubin and Simon Leßenich for discussions about efficient ways to compute types, Łukasz Stafiniak for his help with Toss and Hugo Gimbert for advice on using the Hough transform.

References

Antanas, L.-A.; Thon, I.; van Otterlo, M.; Landwehr, N.; and Raedt, L. D. 2009. Probabilistic logical sequence learning for video. In *Preliminary Proc. of ILP'09*.

Bailey, D. G.; Mercer, K. A.; and Plaw, C. 2004. Autonomous game playing robot. In *Proc. of ICARA'04*.

Barbu, A.; Narayanaswamy, S.; and Siskind, J. M. 2010. Learning physically-instantiated game play through visual observation. In *Proc. of ICRA'10*, 1879–1886.

Canny, J. 1986. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8(6):679–698.

Duda, R. O., and Hart, P. E. 1972. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM* 15(1):11–15.

Etessami, K., and Immerman, N. 2000. Tree canonization and transitive closure. *Inf. Comput.* 157(1-2):2–24.

Fagin, R. 1974. Generalized first-order spectra and polynomial-time recognizable sets. In *SIAM-AMS Proceedings*, volume 7, 27–41.

Gaifman, H. 1982. On local and non-local properties. In *Proc. of the Herbrand Symposium, Logic Colloquium'81*, 105–135. North Holland.

Goodacre, J. 1996. Inductive learning of chess rules using Progol. MSc thesis, Programming Research Group, Oxford.

Grädel, E.; Kolaitis, P. G.; Libkin, L.; Marx, M.; Spencer, J.; Vardi, M. Y.; Venema, Y.; and S.Weinstein. 2007. *Finite Model Theory and Its Applications*. Texts in Theoretical Computer Science. Springer.

Grädel, E. 1999. Why are modal logics so robustly decidable? *Bulletin of the European Association for Theoretical Computer Science* 68:90–103.

Grohe, M. 1998. Fixed-point logics on planar graphs. In *Proc. of LICS'98*, 6–15.

Grohe, M. 1999. Equivalence in finite-variable logics is complete for polynomial time. *Combinatorica* 19(4):507–532.

Grohe, M. 2010. Fixed-point definability and polynomial time on graphs with excluded minors. In *Proc. of LICS'10*, 179–188.

Hanf, W. 1965. Model-theoretic methods in the study of elementary logic. In Addison, J.; Henkin, L.; and Tarski, A., eds., *The Theory of Models*. North Holland. 132–145.

Hazarika, S. M., and Bhowmick, A. 2011. Learning rules of a card game from video. *Artificial Intelligence Review* 1–11.

Immerman, N. 1982. Relational queries computable in polynomial time. In *Proc. of STOC'82*, 147–152.

Immerman, N. 1987. Languages that capture complexity classes. *SIAM J. Comput.* 16(4):760–778.

Kaiser, Ł., and Stafiniak, Ł. 2011. First-order logic with counting for general game playing. In *Proc. of AAAI-11*, 791–796. AAAI Press.

Muggleton, S. 1995. Inverse entailment and Progol. *New Generation Comput.* 13(3&4):245–286.

Needham, C. J.; Santos, P. E.; Magee, D. R.; Devin, V. E.; Hogg, D. C.; and Cohn, A. G. 2005. Protocols from perceptual observations. *Artif. Intell.* 167(1–2):103–136.

Pezzoli, E. 1998. Computational complexity of Ehrenfeucht-Fraïssé games on finite structures. In *Proc. of CSL'98*, 159–170.

Pikhurko, O., and Verbitsky, O. 2010. Logical complexity of graphs: a survey. *CoRR* abs/1003.4865.

Santos, P.; Colton, S.; and Magee, D. R. 2006. Predictive and descriptive approaches to learning game rules from vision data. In *Proc. of IBERAMIA-SBIA*, 349–359.

Toss. `http://toss.sf.net`.

Vardi, M. Y. 1982. The complexity of relational query languages. In *Proc. of STOC'82*, 137–146.