Playgol: learning programs through play

Andrew Cropper
University of Oxford
andrew.cropper@cs.ox.ac.uk

Abstract
Children learn though play. We introduce the analogous idea of learning programs through play. In this approach, a program induction system (the learner) is given a set of tasks and initial background knowledge. Before solving the tasks, the learner enters an unsupervised playing stage where it creates its own tasks to solve, tries to solve them, and saves any solutions (programs) to the background knowledge. After the playing stage is finished, the learner enters the supervised building stage where it tries to solve the user-supplied tasks and can reuse solutions learnt whilst playing. The idea is that playing allows the learner to discover reusable general programs on its own which can then help solve the user-supplied tasks. We claim that playing can improve learning performance. We show that playing can reduce the textual complexity of target concepts which in turn reduces the sample complexity of a learner. We implement our idea in Playgol, a new inductive logic programming system. We experimentally test our claim on two domains: robot planning and real-world string transformations. Our experimental results suggest that playing can substantially improve learning performance. We think that the idea of playing (or, more verbosely, unsupervised bootstrapping for supervised program induction) is an important contribution to the problem of developing program induction approaches that self-discover BK.

1 Introduction
Children learn though play [Schulz et al., 2007; Sim and Xu, 2017; Sim et al., 2017]. We introduce the analogous idea of learning programs through play. In this approach, a program induction system (the learner) is given user-supplied sets of tasks and initial background knowledge (BK). Whereas a standard program induction system would immediately try to solve the tasks, in our approach the learner first enters an unsupervised playing stage. In this stage the learner creates its own tasks to solve, tries to solve them, and saves any solutions (programs) to the BK. After the playing stage is finished, the learner enters the supervised building stage where it tries to solve the user-supplied tasks and can reuse solutions learned whilst playing. The idea is that playing allows the learner to discover reusable general programs on its own which can then be reused in the building stage, and thus improve performance.

To illustrate our play idea, imagine a young child that had never seen Lego before. Suppose you presented the child with Lego bricks and immediately asked them to build a (miniature) home with a pitched roof. The child would probably struggle to build the home without first knowing how to build a solid wall or how to build a pitched roof. Now suppose that before you asked the child to build the home, you first left them alone to play with the Lego. In this scenario the child may start trying to build animals, gardens, ships, or many other seemingly irrelevant things. However, the child is likely to discover reusable and general concepts whilst playing, such as the concept of a stable wall. As we discuss in Section 2, the cognitive science literature shows that children can better learn complex rules after a period of play rather than solely through observation [Schulz et al., 2007; Sim and Xu, 2017; Sim et al., 2017]. In this paper, we explore whether a program induction system can similarly better learn programs after a period of play.

Our idea of using play to discover useful BK contrasts with most forms of program induction which usually require predefined, often human-engineered, static BK as input [Muggleton et al., 2015; Cropper and Muggleton, 2016a; Cropper and Muggleton, 2018; Law et al., 2014; Gulwani, 2011; Kaminski et al., 2018; Evans and Grefenstette, 2018]. Our idea is related to program induction approaches that perform multi-task or meta learning [Lin et al., 2014; Dechter et al., 2013; Ellis et al., 2018a; Ellis and Gulwani, 2017]. In these approaches, a learner acquires useful BK in a supervised manner by solving sets of user-provided tasks, each time saving solutions to the BK, which can then be reused to solve other tasks. In contrast to these supervised approaches, our play approach discovers useful BK in an unsupervised manner whilst playing, where the learner creates its own tasks based on the given BK. Playing can therefore be seen as an unsupervised technique for a learner to discover the BK necessary to solve complex tasks, i.e. a form of unsupervised bootstrapping for supervised program induction.

We claim that playing can improve learning performance. To support this claim, we make the following contributions:
- We introduce the idea of learning programs through play...
2 Related work

Program induction Program induction approaches learn computer programs from input/output examples. Much recent work has focused on task-specific approaches for real-world problems often restricted to specific data types, such as numbers [Singh and Gulwani, 2012] or strings [Gulwani, 2011]. By contrast, we are interested in general program induction approaches. Specifically, we want to develop program induction techniques that discover reusable general concepts, which was the goal of many early AI systems, such as Lenant’s AM system [Lenat, 1977].

Meta-program induction In contrast to universal induction methods [Levin, 1973], program induction approaches use BK as a form of inductive bias [Mitchell, 1997] to restrict the hypothesis space. Most approaches [Muggleton et al., 2015; Cropper and Muggleton, 2016a; Cropper and Muggleton, 2018; Cropper and Muggleton, 2015; Law et al., 2014; Gulwani, 2011; Evans and Grefenstette, 2018; Kaminski et al., 2018; Ellis et al., 2018b; Schüller and Benz, 2018] require as input a fixed, often hand-engineered, BK. To overcome this limitation, several approaches attempt to acquire BK over time [Lin et al., 2014; Dechter et al., 2013; Ellis et al., 2018a; Ellis and Gulwani, 2017], which can be seen as a form of meta-learning [Thrun and Pratt, 2012]. In ILP, meta-learning, also known as multitask learning and automatic bias-revision [Dietterich et al., 2008], involves saving learned programs to the BK so that they can be reused to help learn programs for unsolved tasks. Curriculum learning [Bengio et al., 2009] is a similar idea but requires an ordering over the given tasks. By contrast, our approach, and the aforementioned approaches, do not require an ordering over the tasks.

Lin et al. [Lin et al., 2014] used a technique called dependent learning to enable the MIL system Metagol [Cropper and Muggleton, 2016b] to learn string transformations programs over time. Their approach uses predicate invention to reform the bias of the learner where after a solution is learned not only is the target predicate added to the BK but also its constituent invented predicates. The authors show that their dependent learning approach performs substantially better than an independent (single-task) approach. Dechter et al. [Dechter et al., 2013] studied a similar approach for learning functional programs.

These existing approaches perform supervised meta-learning, i.e. they need a corpus of user-supplied training tasks. By contrast, the main novelty of our playing approach is to perform unsupervised meta-learning where the tasks come not from the user but from the system itself. In other words, our approach allows a learner to discover highly reusable concepts without a user-supplied corpus of training tasks, which Ellis et al. [Ellis et al., 2018a] argue is essential for program induction to become a standard part of the AI toolkit.

Playing Several studies have shown that children learn successfully when they have the opportunity to choose what they want to do. Schulz et al. [Schulz et al., 2007] found that children were able to use self-generated evidence to learn about a causal systems. Sim and Xu [Sim and Xu, 2017] found that three-year-olds were capable of forming higher-order generalisations about a causal system after a short play period. Sim et al. [Sim et al., 2017] showed that children perform significantly better when learning complex clausal rules through free play or by first engaging in free play and then observing, as opposed to solely through observation. As far as we are aware, there is no research studying whether playing can improve machine learning performance, especially in program induction.

Meta-interpretive learning Our idea of learning programs through play is sufficiently general to work with any form of program induction, such as inducing functional programs. However, to clearly explain our theoretical and empirical results, we formalise the problem in an ILP setting [Raedt, 2008] using meta-interpretive learning (MIL) [Muggleton et al., 2015; Cropper and Muggleton, 2016a]. We use MIL for two key reasons. First, MIL supports learning explicitly recursive programs, which is important in the string transformation experiments. Second, MIL uses automatic predicate invention\(^1\) to decompose problems into smaller problems which can then be reused [Cropper and Muggleton, 2016a].

3 Problem setting

We now describe the learning programs through play problem, which, for conciseness, we refer to as the Playgol problem.

3.1 Problem definition

Given a set of tasks and BK, our problem is to induce a set of programs to solve each task. We formalise the problem in an ILP learning from entailment setting [Raedt, 2008]. We assume a language of examples $E$, formed of function-free ground atoms, and languages of background knowledge $B$ and hypotheses $H$, both formed of function-free logic programs. We define the input to the problem:

**Definition 1 (Playgol input).** A Playgol input is a pair $(T, B)$ where:

- $T$ is set of $k$ tasks $\{E_1, E_2, \ldots, E_k\}$, where each $E_i$ is a pair $(E_i^+, E_i^-)$ where $E_i^+ \subseteq E$ and $E_i^- \subseteq E$ represent positive and negative examples respectively of a target predicate
- $B \subseteq B$ is background knowledge

The Playgol problem is to find a consistent program for each task:

\(^1\)Automatic predicate invention contrasts to prescriptive predicate invention where the schema of new predicates (i.e. the arity, and argument types) must be specified by a user.
**Definition 2 (Playgol problem).** Given a Playgol input \((T, B)\), the goal is to return a set of hypotheses \(\{H_i \in \mathcal{H}(E^+_i, E^-_i) \in T, (H_i \cup B \models E^+_i) \land (H_i \cup B \not\models E^-_i)\}\)  

### 3.2 Meta-interpretable learning

We solve the Playgol problem using MIL, a form of ILP based on a Prolog meta-interpreter. For brevity, we omit a formal description of MIL, and refer the reader to the literature for more details [Cropper and Muggleton, 2016a]. We instead provide an informal overview. A MIL learner is given as input sets of atoms representing positive and negative examples of a target concept, background knowledge in the form of a logic program, and, crucially, a set of second-order formulas called metarules. A MIL learner works by trying to construct a proof of the positive examples. It uses the metarules to guide the proof search. Metarules can therefore be seen as program templates. Figure 1 shows some commonly used metarules. Once a proof is found a MIL learner extracts a logic program, and, crucially, a set of second-order formulas called metarules. Once a proof is found a MIL learner extracts a logic program from the proof and checks that it is inconsistent with the negative examples. If not, it backtracks to consider alternative proofs.

| Name | Metarule |
|------|----------|
| precon | \(P(A, B) \leftarrow Q(A), R(A, B)\) |
| postcon | \(P(A, B) \leftarrow Q(A, B), R(B)\) |
| chain | \(P(A, B) \leftarrow Q(A, C), R(C, B)\) |
| tailrec | \(P(A, B) \leftarrow Q(A, C), P(C, B)\) |

Figure 1: Example metarules. The letters \(P\), \(Q\), and \(R\) denote existentially quantified variables. The letters \(A\), \(B\), and \(C\) denote universally quantified variables.

### 3.3 Sample complexity

We claim that playing can improve learning performance. We support this claim by showing that playing can reduce the size of the MIL hypothesis space which in turn reduces sample complexity and expected error. In MIL, the size of the hypothesis space is a function of the metarules, the number of background predicates, and the maximum program size. We restrict metarules by their body size and literal arity:

**Definition 3.** A metarule is in \(\mathcal{M}_j\) if it has at most \(j\) literals in the body and each literal has arity at most \(i\).

By restricting the form of metarules we can calculate the size of a MIL hypothesis space:

**Proposition 1 (Hypothesis space [Cropper and Tourret, 2018]).** Given \(p\) predicate symbols and \(m\) metarules in \(\mathcal{M}_j\), the number of programs expressible with \(n\) clauses is \((mp^j + 1)^n\).

We use this result to show the MIL sample complexity:

**Proposition 2 (Sample complexity [Cropper and Tourret, 2018]).** Given \(p\) predicate symbols, \(m\) metarules in \(\mathcal{M}_j\), and a clause bound \(n\), MIL has sample complexity \(s\) with error \(\epsilon\) and confidence \(\delta\):

\[
s \geq \frac{1}{\epsilon}(n \ln(n) + (j + 1)n \ln(p) + \ln \frac{1}{\delta})
\]

Proposition 2 helps explain our idea of playing. When playing, a learner creates its own tasks and saves any solutions to the BK, which increases the number of predicate symbols \(p\). The solutions learned whilst playing may in turn help solve the user-supplied tasks, i.e. could reduce the size \(n\) of the target program. For example, if trying to learn sorting algorithms, a learner could discover the concepts of partition and merge when playing which could then help learn quicksort and mergesort respectively. In other words, the key idea of playing is to increase the number of predicate symbols \(p\) in order to reduce the size \(n\) of the target program. We consider when playing can reduce sample complexity:

**Theorem 1 (Playgol improvement).** Given \(p\) predicate symbols and \(m\) metarules in \(\mathcal{M}_j\), let \(n\) be the minimum numbers of clauses needed to express a target theory with standard MIL. Let \(n - k\) be the minimum numbers of clauses needed to express a target theory with Playgol using an additional \(c\) predicate symbols. Let \(s\) and \(s'\) be the bounds on the number of training examples required to achieve error less than \(\epsilon\) with probability at least \(1 - \delta\) with standard MIL and Playgol respectively. Then \(s > s'\) when:

\[
n \ln(p) > (n - k) \ln(p + c)
\]

**Proof.** Follows from Proposition 2 and rearranging of terms. \(\square\)

Theorem 1 shows when playing can reduce sample complexity compared to not playing. In such cases, if the number of training examples is fixed for both approaches, the corresponding discrepancy in sample complexity is balanced by an increase in predictive error [Blumer et al., 1989]. In other words, Theorem 1 shows that adding extra (sometimes irrelevant) predicates to BK can improve learning performance so long as some can be reused to learn new programs. Playgol implements this idea.

### 4 Playgol

Algorithm 1 shows the Playgol algorithm, which uses Metagol [Cropper and Muggleton, 2016b], a MIL implementation, as the main learning algorithm. Playgol takes as input a set of user-supplied build tasks \(T_b\), initial background knowledge BK, and a maximum search depth \(\max_d\). Playgol first enters the unsupervised playing stage. In this stage, Playgol creates its own play tasks \(T_p\) by sampling elements from the instance space. Playgol then uses a dependent learning approach [Lin et al., 2014] to expand the BK. Starting at depth \(d=1\), Playgol tries to solve each play task using at most \(d\) clauses. To solve an individual task, Playgol calls Metagol. Each time a play task is solved, the solution (program) is added the BK and can be reused to help solve other play tasks. Once Playgol has tried to solve all play tasks at depth \(d\), it increases the depth and tries to solve the remaining play tasks. Playgol repeats this process until it reaches a maximum depth \((\max_d)\), then it returns the initial BK augmented with solutions to the play tasks. Playgol then enters the building stage, in which it tries to solve each user-supplied build task using the augmented BK using a standard independent learning approach, eventually returning a set of induced programs.
We claim that playing can improve learning performance. We now experimentally test our claim. We test the null hypothesis:

**Null hypothesis 1** Playing cannot improve learning performance.

Theorem 1 shows that playing can reduce sample complexity compared to not playing. Theorem 1 does not, however, state how many play tasks are needed to improve learning performance. Playgol creates its own play tasks by sampling from the instance space. Suppose we sampled uniformly at random without replacement from a finite instance space. Then if we sample enough times we will sample every instance. One could therefore argue that Playgol is doing nothing more than sampling play tasks that it will eventually have to solve (i.e., Playgol is sampling build tasks whilst playing). To refute this argument we test the null hypothesis:

**Null hypothesis 2** Playing cannot improve learning performance without any play tasks.

To test null hypotheses 1 and 2 we compare Playgol’s performance when varying the number of play tasks. When there are no play tasks Playgol is equivalent to Metagol.

A key motivation for using MIL is that it supports predicate invention. Although we provide no theoretical justification, we claim that predicate invention is useful when playing because it allows for problems to be decomposed into smaller reusable sub-problems. We test this claim with the null hypothesis:

**Null hypothesis 3** Saving invented predicates whilst playing cannot improve learning performance.

To test null hypothesis 3 we also use a version of Playgol which does not save invented predicates to the BK, which we call Playgol_{nopi}.

### 5.1 Robot planning

Our first experiment focuses on learning robot plans.

#### Algorithm 1 Playgol

```plaintext
func playgol(T_b, BK, max_d)
    BK = play(BK, max_d)
    return build(T_b, BK, max_d)

func play(BK, max_d)
    T_p = gen_tasks(BK)
    for d=1 to max_d
        for (E^+, E^-) in T_p
            prog = metagol(BK, E^+, E^-, max_d)
        if prog != null
            BK = BK \ {prog}
    T_p = T_p \ {(E^+, E^-)}
    return BK

func build(T_b, BK, max_d)
    P = {}
    for (E^+, E^-) in T_b
        prog = metagol(BK, E^+, E^-, max_d)
    if prog != null
        P = P \ {prog}
    return P
```

#### Materials

There is a robot and a ball in an $n^2$ space. The robot can move around and can grab and drop the ball. Figure 2 shows example initial and final states. The goal is to learn a program to move from the initial state to the final state. The robot can perform six dyadic actions to transform the state: up, down, right, left, grab, and drop. Training examples are atoms of the form $f(s_1, s_2)$, where $f$ is the target predicate and $s_1$ and $s_2$ are initial and final states respectively. Figure 3 shows an example solution (a program and the corresponding plan) for Figure 2. We allow Playgol to learn programs using the ident and chain metarules (Figure 1). We use $5^2$ and $6^2$ spaces with instance spaces $X_5$ and $X_6$ respectively. The instance spaces contain all possible $f(s_1, s_2)$ atoms. The cardinalities of $X_5$ and $X_6$ are approximately $5^8$ and $6^8$ respectively.

![Figure 2: Robot planning example.](image)

![Figure 3: Robot planning solution, where (a) is a Prolog program and (b) is the plan that the program executes. The predicate f1 is an invented predicate.](image)
(c) Measure the percentage of correct solutions in $P_p$.
We enforce a timeout of 60 seconds per play and build task.
We measure the standard error of the mean over 10 repetitions.

**Results** Figure 4 shows that Playgol solves more build tasks
given more play tasks. For the $5^2$ space, Playgol solves only
12% of the build tasks without playing. The baseline represents
the performance of Metagol (i.e. learning without play). By contrast,
playing improves performance in all cases. After
1000 play tasks, Playgol solves almost 100% of the build
tasks. For the $6^2$ space, the results are similar, where the build
performance is only 7% without playing but over 60% after
1200 play tasks. These results suggest that we can reject null
hypothesis 1, i.e. we can conclude that playing can improve
learning performance.

As already mentioned, one may argue that Playgol is simply
sampling build tasks as play tasks. Such duplication may occur.
In this experiment, for us to sample all of the build tasks
we would expect to sample $\Theta(|X_n| \log(|X_n|))$ play tasks$^3$,
which corresponds to sampling approximately 5 million and 24
million tasks for the $5^2$ and $6^2$ spaces respectively. However,
our experimental results show that to solve almost all of the
build tasks we only need to sample approximately 1000 and
2000 play tasks for the $5^2$ and $6^2$ spaces respectively. These
values are less than 1/1000 of the expected rate. Therefore, our
experimental results suggest that we can reject null hypothesis
2, i.e. we can conclude that playing can improve learning performance
without needing to sample many play tasks.

Finally, Figure 4 shows that Playgol solves more tasks than
Playgol$_{nopi}$, although in the $5^2$ space both approaches con-
verge after 2000 play tasks. A McNemar’s test on the results
of Playgol and Playgol$_{nopi}$ confirmed the significance at the
$p < 0.001$ level for the $5^2$ and $6^2$ spaces respectively. This result suggests
that we can reject null hypothesis 3, i.e. we can conclude that
predicate invention can improve learning performance when playing.

![Figure 4: Robot experiment results. The baseline represents learning without play (i.e. standard Metagol).](image)

(a) $5^2$ space  
(b) $6^2$ space

5.2 String transformations

Our first experiment tested the null hypotheses in a controlled
experimental setting. We now see whether playing can im-
prove learning performance on ‘real-world’ string transformations.

**Materials** We use 94 real-word string transformation tasks.
Each task has 10 examples. Each example is an atom of
the form $f(x, y)$ where $f$ is the task name and $x$ and $y$ are
input and output strings respectively. Figure 5 shows three examples
for the task build$_{95}$, where the goal is to learn a program that extracts the first three letters of the month
name and makes them uppercase.

| Input                     | Output |
|---------------------------|--------|
| 22 July,1983 (35 years old) | JUL |
| 30 October,1955 (63 years old) | OCT |
| 2 November,1954 (64 years old) | NOV |

![Figure 5: Examples for the build$_{95}$ string transformation problem.](image)

In the build stage we use the real-word tasks. In the play
stage, Playgol samples play tasks from the instance space
formed of random string transformations. The play tasks are
formed from an alphabet with 80 symbols, including the letters
a-z, A-Z, the numbers 0-9, and punctuation symbols (’,’:;",+,-
...,etc). To generate a play task we use the procedure:

1. Select a random integer $l$ between 3 and 20 to represent the
   input length
2. Generate a random string $x$ of length $l$ to represent the input
   string
3. Select a random integer $p$ between 3 and 20 and enumerate
   all programs $P$ of length $p$ consistent with $x$
4. Select a random program from $P$ and apply it to $x$ to generate
   the output string $y$ to form the example $f(x, y)$ where
   $f$ is the play task name

This procedure only generates play tasks that are theoretically
solvable, i.e. for which there is a hypothesis in the hypothesis
space. In other words, it generates play tasks based on the
given BK. Figure 6 shows example play tasks.

| Task  | Input       | Output |
|-------|-------------|--------|
| play$_9$ | .I/(3, R) | F     |
| play$_{52}$ | @B4, X;3MjKdy2zC | B |
| play$_{136}$ | 9pF*$ktbf51v | 99PF |
| play$_{228}$ | 1IzhiQk- | Q     |

![Figure 6: Examples of randomly generated play tasks for the string transformation experiment.](image)

The play instance space $X$ contains all possible string trans-
formations consistent with the aforementioned procedure. The
space contains approximately $80^{40}$ atoms$^4$.

We provide Playgol with the metarules precon, postcon,
chain, and tailrec; the monadic predicates: empty, space,
letter, number, uppercase, lowercase; the negations of the
monadics not_empty, not_space, etc; and the dyadic predicates
copy, skip, mk_uppercase, mk_lowercase.

$^3$This problem is an instance of the coupon collectors problem [Wikipedia contributors, 2018]

$^4$In the case that the input is length 20 there are $80^{20}$ possible strings, thus $80^{40}$ pairs.
Method  Our experimental method is:

1. For each set of user-supplied build tasks $T_i$, sample uniformly without replacement 5 atoms from $T_i$ to form the build training examples $T_{i,train}$ and use the other 5 atoms as the build testing examples $T_{i,test}$.
2. For each $p$ in $\{0, 200, 400, \ldots, 2000\}$:
   
   (a) Sample uniformly with replacement $p$ atoms from $X$ to form the play tasks $T_{i,p,play}$ (this step corresponds to gen_tasks(BK) in Algorithm 1)
   
   (b) Call $\text{playgol}(BK, T_{i,train}, 5)$ which returns a set of programs $P_{i,p}$
   
   (c) Measure the predictive accuracy of $P_{i,p}$ against the testing examples $T_{i,test}$

We enforce a learning timeout of 60 seconds per play and build task. If Playgol learns no program then every test example is deemed false. We measure the standard error of the mean over 10 repetitions.

Results  Figure 7 shows the mean predictive accuracies of Playgol when varying the number of play tasks. Note that we are not interested in the absolute predictive accuracies of Playgol, which are low because of the small timeout and also the difficulty of the problems. We are instead interested in how the accuracies change given more play tasks, and also the difference in accuracies between Playgol and Playgol$_{nopi}$. Figure 7 shows that Playgol’s predictive accuracy improves given more play tasks. Playgol’s accuracy is 25% without playing. By contrast, playing improves accuracy in all cases. After 2000 play tasks, the accuracy is almost 37%, an improvement of 12%.

Figure 8 shows an example of when playing improved building performance, where the solution to the build task b224 is composed of the solutions to many play tasks. The solutions to the play tasks are themselves are often composed of solutions to other play tasks, including reusing many invented predicates. This example clearly demonstrates the use of predicate invention to discover highly reusable concepts that build on each other.

Overall the results from this experiment add further evidence for rejecting all the null hypotheses.

6 Conclusions and future work

We have introduced the idea of learning programs through play. In this approach, a program induction system creates its own tasks to solve, tries to solve them, and saves any solutions to its BK, which can then be reused to solve the user-supplied tasks. The main novelty of our playing approach is to perform unsupervised meta-learning where the learner creates its own tasks. By contrast, existing meta-learning approaches are supervised and need a corpus of user-supplied training tasks. We claimed that playing can improve learning performance. Our theoretical results support this claim and show that playing can reduce the sample complexity of a learner (Theorem 1). We have implemented our idea in Playgol, a new ILP system. Our experimental results on two domains (robot planning and string transformations) further support our claim and show that playing can substantially improve learning performance without the need for many play tasks. Our experimental results also show that predicate invention can improve learning performance because it allows a learner to discover highly reusable sub-programs.

6.1 Limitations and future work

Which domains?  We have demonstrated the benefits of playing on robot planning and string transformations. However, the generality of the approach is unclear. Theorem 1 shows conditions for when playing can reduce sample complexity and helps explain our empirical results. Theorem 1 does not, however, identify a priori on which domains playing is useful. Our preliminary work suggests that playing is useful in other domains, including when inducing graphics programs where playing discovers general concepts such as a vertical or horizontal line. Future work should determine in which domains playing is useful.

How many play tasks?  Our robot experiments show that as the instance space grows Playgol needs to sample more tasks to achieve high performance. In future work we want to develop a theory that predicts how many play tasks Playgol needs to sample to substantially improve learning performance.

Better sampling  In the string transformation experiment playing did not continue to improve performance as it did in the robot experiment. One explanation for this performance plateau is the relevancy of sampled play tasks. In the robot experiment we sampled play and build tasks from the same distribution. By contrast, in the string transformation experiment,
we sampled random play tasks but used real-world build tasks. In future work, we want to explore methods to sample more useful play tasks. For instance, rather than create play tasks in an unsupervised manner, it may be beneficial to create play tasks similar to build tasks, i.e. in a semi-supervised manner.

**Forgetting** Proposition 1 shows that the size of the MIL hypothesis space is a function of the size of target program and also the number of background predicates. In our playing approach the learner saves all the solutions for the play tasks to the BK. This approach is clearly non-optimal because not all of the solutions will be reusable. In future work we want to develop techniques to determine which solutions learned during playing would be beneficial to help solve the build tasks and to forget the rest.

**Summary** We have shown that playing can substantially improve learning performance. We think that the idea of playing (or more verbosely unsupervised bootstrapping for supervised program induction) opens an exciting research area focusing on how program induction systems can discover their own BK without the need for user-supplied tasks.

**References**

[Bengio et al., 2009] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In ICML 2009, volume 328 of ACM International Conference Proceeding Series, pages 41–48. ACM, 2009.

[Blumer et al., 1989] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Learnability and the vapnik- chervonenkis dimension. J. ACM, 36(4):929–965, 1989.

[Cropper and Muggleton, 2015] Andrew Cropper and Stephen H. Muggleton. Learning efficient logical robot strategies involving composable objects. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, pages 3423–3429. AAAI Press, 2015.

[Cropper and Muggleton, 2016a] Andrew Cropper and Stephen H. Muggleton. Learning higher-order logic programs through abstraction and invention. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016, pages 1418–1424. IJCAI/AAAI Press, 2016.

[Cropper and Muggleton, 2016b] Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.

[Cropper and Muggleton, 2018] Andrew Cropper and Stephen H. Muggleton. Learning efficient logic programs. Machine Learning, Apr 2018.

[Cropper and Tourret, 2018] Andrew Cropper and Sophie Tourret. Derivation reduction of metarules in meta-interpretive learning. In ILP, volume 11105 of Lecture Notes in Computer Science, pages 1–21. Springer, 2018.

[Dechter et al., 2013] Eyal Dechter, Jonathan Malmud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In Francesca Rossi, editor, IJCAI 2013, pages 1302–1309. IJCAI/AAAI, 2013.

[Dietterich et al., 2008] Thomas G. Dietterich, Pedro M. Domingos, Lise Getoor, Stephen Muggleton, and Prasad Tadepalli. Structured machine learning: the next ten years. Machine Learning, 73(1):3–23, 2008.

[Ellis and Gulwani, 2017] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, pages 1638–1645. ijcai.org, 2017.

[Ellis et al., 2018a] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program induction. In Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada., pages 7816–7826, 2018.

[Ellis et al., 2018b] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada., pages 6602–6611, 2018.

[Evans and Grefenstette, 2018] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. J. Artif. Intell. Res., 61:1–64, 2018.

[Gulwani, 2011] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 317–330, 2011.

[Kaminski et al., 2018] Tobias Kaminski, Eyal Dechter, and Sumit Gulwani. Exploiting answer set programming with external sources for meta-interpretive learning. TPLP, 18(3-4):571–588, 2018.

[Law et al., 2014] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In JELIA 2014, pages 311–325, 2014.

[Lenat, 1977] Douglas B. Lenat. Automated theory formation in mathematics. In Raj Reddy, editor, Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977, pages 833–842. William Kaufmann, 1977.

[Levin, 1973] Leonid Anatolevich Levin. Universal sequential search problems. Problemy Peredachi Informatsii, 9(3):115–116, 1973.

[Lin et al., 2014] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In ECAI 2014, pages 525–530, 2014.

[Mitchell, 1997] Tom M. Mitchell. Machine learning. McGraw Hill series in computer science. McGraw-Hill, 1997.

[Muggleton et al., 2014] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. Machine Learning, 94(1):49–73, 2014.

[Muggleton et al., 2015] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. Machine Learning, 100(1):49–73, 2015.

[Raedt, 2008] Luc De Raedt. Logical and relational learning. Cognitive Technologies. Springer, 2008.

[Schüller and Benz, 2018] Peter Schüller and Mishal Benz. Best-effort inductive logic programming via fine-grained cost-based hypothesis generation - the inspire system at the inductive logic.
programming competition. *Machine Learning*, 107(7):1141–1169, 2018.

[Schulz et al., 2007] Laura E. Schulz, Alison Gopnik, and Clark Glymour. Preschool children learn about causal structure from conditional interventions. *Developmental science*, 10(3):322–332, 2007.

[Sim and Xu, 2017] Zi L. Sim and Fei Xu. Learning higher-order generalizations through free play: Evidence from 2-and 3-year-old children. *Developmental psychology*, 53(4):642, 2017.

[Sim et al., 2017] Zi Lin Sim, Kuldeep K. Mahal, and Fei Xu. Is play better than direct instruction? learning about causal systems through play. In *Proceedings of the 39th Annual Meeting of the Cognitive Science Society, CogSci 2017, London, UK, 16-29 July 2017*. cognitivesciencesociety.org, 2017.

[Singh and Gulwani, 2012] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 634–651, 2012.

[Thrun and Pratt, 2012] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

[Wikipedia contributors, 2018] Wikipedia contributors. Coupon collector's problem — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-October-2018].