Forgetting to Learn Logic Programs

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

Abstract

Most program induction approaches require predefined, often hand-engineered, background knowledge (BK). To overcome this limitation, we explore methods to automatically acquire BK through multi-task learning. In this approach, a learner adds learned programs to its BK so that they can be reused to help learn other programs. To improve learning performance, we explore the idea of *forgetting*, where a learner can additionally remove programs from its BK. We consider forgetting in an inductive logic programming (ILP) setting. We show that forgetting can significantly reduce both the size of the hypothesis space and the sample complexity of an ILP learner. We introduce *Forgetgol*, a multi-task ILP learner which supports forgetting. We experimentally compare Forgetgol against approaches that either remember or forget everything. Our experimental results show that Forgetgol outperforms the alternative approaches when learning from over 10,000 tasks.

1 Introduction

A major challenge in machine learning is inductive bias: how to choose a learner's hypothesis space so that it is large enough to contain the target hypothesis, yet small enough to be searched efficiently (Baxter 2000). A major challenge in program induction is how to chose a learner's background knowledge (BK), which in turn defines the hypothesis space. Most approaches need predefined, often hand-engineered, BK (Law, Russo, and Broda 2014; Balog et al. 2017; Evans and Grefenstette 2018; Ellis et al. 2018b). By contrast, we want a learner to automatically learn BK.

Several approaches (Dechter et al. 2013; Ellis et al. 2018a; Cropper 2019) learn BK through *meta-learning*. The idea is to use knowledge gained from solving one problem to help solve a different problem. For instance, Lin et al. (2014) use this approach on 17 string transformation problems. They automatically identify easier problems, learn programs for them, and then reuse the learned programs to help learn programs for more difficult problems. The authors experimentally show that their multi-task approach performs substantially better than a single-task approach because learned programs are frequently reused. They also show that this approach leads to a hierarchical library of reusable programs.

However, no approach has been shown to work on many (>10,000) tasks. Moreover, most approaches struggle even on few tasks (Quinlan 1990; Ferri, Hernández-Orallo, and Ramírez-Quintana 2001). By contrast, humans easily learn thousands of diverse concepts. If we want to develop human-like AI systems that support lifelong learning – a grand challenge in AI (Silver, Yang, and Li 2013; Lake et al. 2017) – then we need to overcome this limitation.

We think the key limitation with existing approaches is that they save all learned programs to the BK, which is a problem because too much irrelevant BK is detrimental to learning performance (Srinivasan, Muggleton, and King 1995; Srinivasan, King, and Bain 2003). The Blumer bound (Blumer et al. 1987)[1] helps explain this problem. This bound states that given two hypothesis spaces, searching the smaller space will result in fewer errors compared to the larger space, assuming that the target hypothesis is in both spaces. Here lies the problem: how to choose a learner's hypothesis space so that it is large enough to contain the target hypothesis yet small enough to be efficiently searched.

To address this problem, we explore the idea of *forgetting*. In this approach, a learner continually grows and shrinks its hypothesis space by adding and removing programs to and from its BK. We claim that forgetting can improve learning performance, especially when learning from many tasks. To support this claim, we make the following contributions:

- We define forgetting in an inductive logic programming (ILP) setting (Section 3). We show that forgetting can reduce both the size of the hypothesis space (Theorem 2) and the sample complexity of an ILP learner (Theorem 3). We show that optimal forgetting is NP-Hard (Theorem 1).
- We introduce *Forgetgol*, a multi-task ILP learner that continually learns, saves, and forgets programs (Section 4). We introduce two forgetting methods: *syntactical* and *statistical* forgetting, the latter based on Theorem 2.
- We experimentally show on two datasets (robots and Lego) that forgetting can substantially improve learning performance when learning from over 10,000 tasks (Section 5). This experiment is the first to consider the performance of a learner on many tasks.

[1] The bound is a reformulation of Lemma 2.1.

2 Related Work

Several approaches (Dumancic and Blockeel 2017; Dumancic et al. 2019; Cropper 2019) learn BK though unsupervised learning. These approaches do not require user-supplied tasks as input. By contrast, we learn BK through supervised learning and need a corpus of tasks as input. Our forgetting idea is, however, equally applicable to unsupervised approaches.

Supervised approaches perform either *multi-task* (Caruana 1997) or *lifelong* (Silver, Yang, and Li 2013) learning. A multi-task learner is given a *set* of tasks and can solve them in any order. A lifelong learner is given a *sequence* of tasks and can use knowledge gained from the past n-1 tasks to help solve the nth task. The idea behind both approaches is to reuse knowledge gained from solving one problem to help to solve a different problem, i.e. perform *transfer learning* (Torrey and Shavlik 2010). Although we focus on multi-task learning, our forgetting idea is suitable for both approaches.

Lin et al. (2014) state that an important problem with multi-task learning is how to handle the complexity of growing BK. To address this problem, we propose forgetting, where a learner continually revises its BK by adding and removing programs, i.e. performs automatic bias-revision (Dietterich et al. 2008). Forgetting has long been recognised as an important feature in AI (Lin and Reiter 1994). Most forgetting work focuses on eliminating logical redundancy (Plotkin 1971), e.g. to improve efficiency in SAT (Heule et al. 2015). Our approach is slightly unusual because we are willing to forget logically irredundant knowledge.

Our forgetting idea comes from the observation that existing approaches remember everything (Quinlan 1990; Ferri, Hernández-Orallo, and Ramírez-Quintana 2001; Lin et al. 2014; Dechter et al. 2013; Cropper 2019), which is a problem because too much irrelevant BK is detrimental to learning performance (Srinivasan, Muggleton, and King 1995; Srinivasan, King, and Bain 2003).

A notable case of forgetting in machine learning is *catastrophic forgetting* (McCloskey and Cohen 1989), the tendency of neural networks to forget previously learned knowledge upon learning new knowledge. In multi-task program induction we have the inverse problem: *catastrophic remembering*, the inability for a learner to forget knowledge. We therefore need *intentional forgetting* (Beierle and Timm 2019).

Forgetting in program induction mostly focuses on forgetting examples (Sablon and Raedt 1995; Widmer and Kubat 1996; Maloof 1997). By contrast we forget BK. Martínez-Plumed et al. (2015) explore ways to compress BK without losing knowledge. Our work differs in many ways, mainly because we work in a multi-task setting with many tasks. Forgetting is essentially identifying irrelevant BK. Deepcoder (Balog et al. 2017) and Dreamcoder (Ellis et al. 2018a) both train neural networks to score how relevant programs are in the BK. Whereas Deepcoder's BK is fixed and predetermined by the user, we grow and revise our BK through multi-task learning. Dreamcoder also revises its BK through multi-task learning. Our work differs from Dreamcoder because we (1) formally show that forgetting can improve learning performance, (2) describe forgetting methods that do not require neural networks, and (3) learn from 20,000 tasks, whereas Dreamcoder considers at most 236 tasks.

3 Problem Setting

We now define the forgetting problem, show that optimal forgetting is NP-hard, and that forgetting can reduce both the size of the hypothesis space and the sample complexity of a learner.

3.1 ILP Problem

We base our problem on the ILP learning from entailment setting (Raedt 2008). We assume standard logic programming definitions throughout (Lloyd 2012). We define the input:

Definition 1 (**ILP input**). An ILP input is a tuple (B, E^+, E^-) where B is logic program background knowledge, and E^+ and E^- are sets of ground atoms which represent positive and negative examples respectively.

By *logic program* we mean a set of definite clauses, and thus any subsequent reference to a clause refers to a definite clause. We assume that B contains the necessary language bias, such as mode declarations (Muggleton 1995) or metarules (Cropper and Tourret 2019), to define the hypothesis space:

Definition 2 (**Hypothesis space**). The hypothesis space $\mathcal{H}_B$ is the set of all hypotheses (logic programs) defined by background knowledge B.

The version space contains all complete (covers all the positive examples) and consistent (covers none of the negative examples) hypotheses:

Definition 3 (**Version space**). Let (B, E^+, E^-) be an ILP input. Then the version space is $\mathcal{V}_{B,E^+,E^-} = \{H \in \mathcal{H}_B | (H \cup B \models E^+) \wedge (\forall e^- \in E^-, H \cup B \not\models e^-)\}$.

We define an optimal hypothesis:

Definition 4 (**Optimal hypothesis**). Let $I = (B, E^+, E^-)$ be an ILP input and $cost : \mathcal{H} \rightarrow R$ be an arbitrary cost function. Then $H \in \mathcal{V}_{B,E^+,E^-}$ is an *optimal hypothesis* for I if and only if $cost(H) \leq cost(H')$ for all $H' \in \mathcal{V}_{B,E^+,E^-}$.

The cost of a hypothesis could be many things, such as the number of literals in it (Law, Russo, and Broda 2014) or its computational complexity (Cropper and Muggleton 2019). In this paper, the cost of a hypothesis is the number of clauses in it.

The ILP problem is to find a complete and consistent hypothesis:

Definition 5 (**ILP problem**). Given an ILP input (B, E^+, E^-), the ILP problem is to return a hypothesis $H \in \mathcal{V}_{B,E^+,E^-}$.

A learner should ideally return the optimal hypothesis, but it is not an absolute requirement, and it is common to sacrifice optimality for efficiency.

3.2 Forgetting

The forgetting problem is to find a subset of the given BK from which one can still learn the optimal hypothesis:

Definition 6 (**Forgetting problem**). Let $I = (B, E^+, E^-)$ be an ILP input and H be an optimal hypothesis for I. Then the *forgetting problem* is to return $B' \subset B$ such that $H \in \mathcal{V}_{B',E^+,E^-}$. We say that B' is *reduced* background knowledge for E^+ and E^-.

We would ideally perform optimal forgetting:

Definition 7 (**Optimal forgetting**)**.** Let $I = (B, E^+, E^-)$ be an ILP input and H be an optimal hypothesis for I. Then the *optimal forgetting* problem is to return $B' \subset B$ such that $H \in \mathcal{V}_{B',E^+,E^-}$ and there is no $B'' \subset B'$ such that $H \in \mathcal{V}_{B'',E^+,E^-}$.

Even if we assume that the forgetting problem is decidable, optimal forgetting is NP-hard:

Theorem 1 (**Complexity**)**.** Optimal forgetting is NP-Hard.

Proof. We can reduce the knapsack problem, which is NP-Hard, to optimal forgetting. □

3.3 Meta-Interpretive Learning

We now show the benefits of forgetting in meta-interpretive (MIL) (Muggleton, Lin, and Tamaddoni-Nezhad 2015; Cropper, Morel, and Muggleton 2019), a powerful form of ILP that supports predicate invention (Stahl 1995) and learning recursive programs. For brevity, we omit a formal description of MIL and instead provide a brief overview. A MIL learner is given as input two sets of ground atoms that represent positive and negative examples of a target concept, BK described as a logic program, and a set of higher-order Horn clauses called metarules (Table 1). A MIL learner uses the metarules to construct a proof of the positive examples and none of the negative examples, and forms a hypothesis using the substitutions for the variables in the metarules. For instance, given positive and negative examples of the *grandparent* relation, background knowledge with the *parent* relation, and the metarules in Table 1, a MIL learner could use the *chain* metarule with the substitutions {P/grandparent, Q/parent, R/parent} to induce the theory: *grandparent(A,B) ← parent(A,C), parent(C,B).*

Name	**Metarule**
ident	$P(A,B) \leftarrow Q(A,B)$
precon	$P(A,B) \leftarrow Q(A), R(A,B)$
postcon	$P(A,B) \leftarrow Q(A,B), R(B)$
chain	$P(A,B) \leftarrow Q(A,C), R(C,B)$

Table 1: Example metarules. The letters P, Q, and R denote second-order variables. The letters A, B, and C denote first-order variables.

The size of the MIL hypothesis space is a function of the metarules, the number of predicate symbols in the BK, and a target hypothesis size. We define $sig(P)$ to be the set of predicate symbols in the logic program P. In the next two propositions, assume an ILP input with background knowledge B, where $p = |sig(B)|$, m is the number of metarules in B, each metarule in B has at most j body literals, and n is a target hypothesis size. Cropper et al. (2019) show that the size of the MIL hypothesis space is:

Proposition 1 (**Hypothesis space**)**.** The size of the MIL hypothesis space is at most $(mp^{j+1})^n$.

The authors use this result with the Blumer bound to show the MIL sample complexity in a PAC learning setting (Valiant 1984):

Proposition 2 (**Sample complexity**)**.** With error ϵ and confidence δ MIL has sample complexity $s \geq \frac{1}{\epsilon}(n\ln(m) + (j+1)n\ln(p) + \ln\frac{1}{\delta})$.

We use these results in the next section to show that forgetting can reduce sample complexity in MIL and in Section 4.3 to define a statistical forgetting method.

3.4 Forgetting Sample Complexity

The purpose of forgetting is to reduce the size of the hypothesis space without excluding the target hypothesis. In MIL we want to reduce the number of predicate symbols without excluding the target hypothesis from the hypothesis space. We now show the potential benefits of doing so. In the next two theorems, assume an ILP input (B, E^+, E^-) and let $B' \subset B$ be reduced background knowledge for E^+ and E^-, where B and B' both contain m metarules with at most j body literals, $p = |sig(B)|$, $p' = |sig(B')|$, n be a target hypothesis size, and $r = p'/p$, i.e. r is the forgetting *reduction ratio*.

Theorem 2 (**Hypothesis space reduction**)**.** Forgetting reduces the size of the MIL hypothesis space by a factor of $r^{(j+1)n}$.

Proof. Replacing p with rp in Proposition 1 and rearranging terms leads to $r^{(j+1)n}(mp^{(j+1)})^n$. □

Theorem 3 (**Sample complexity reduction**)**.** Forgetting reduces sample complexity in MIL by $(j+1)n\ln(r)$.

Proof. Replacing p with rp in Proposition 2 and rearranging terms leads to $s \geq \frac{1}{\epsilon}(n\ln(m) + (j+1)n\ln(r) + (j+1)n\ln(p) + \ln\frac{1}{\delta})$. □

These results show that forgetting can substantially reduce the size of the hypothesis space and sample complexity of a learner.

4 Forgetgol

We now introduce Forgetgol, a multi-task ILP system which supports forgetting. Forgetgol relies on Metagol (Cropper and Muggleton 2016), a MIL system. We first briefly describe Metagol.

4.1 Metagol

Metagol is based on a Prolog meta-interpreter. Metagol takes as input (1) a logic program which represents BK, (2) a set of predicate symbols S which denotes symbols that may appear in a hypothesis, similar to determinations in Aleph (Srinivasan 2001), (3) two sets of ground atoms which represent positive and negative examples, and (4) a maximum program size. The set S is necessary because the BK may contain relations which should not appear in a hypothesis. For instance, the BK could contain a definition for *quicksort* which uses *partition* and *append* but we may want Metagol to only use *quicksort* in a hypothesis. The set S is usually defined as part of the BK but we separate it for clarity.

Given these inputs, Metagol tries learn a logic program by proving each atom in the set of positive examples. Metagol first tries to prove an atom using the BK by delegating the

Algorithm 1 Forgetgol

```
1   func forgetgol(B,S,T,max$_d$,forget):
2     P = {}
3     for depth=1 to max$_d$:
4       S$'$ = forget(S,B)
5       S$_d$, T$_d$, P$_d$, = learn(B,S$'$,T,depth)
6       B = B $\cup$ P$_d$
7       S = S $\cup$ S$_d$
8       T = T $\setminus$ T$_d$
9       P = P $\cup$ P$_d$
10    return P
11
12  func learn(B,S,T,depth):
13    S$_d$ = {}
14    T$_d$ = {}
15    P$_d$ = {}
16    for (E$^+$,E$^-$) in T:
17      prog = metagol(B,S,E$^+$,E$^-$,depth)
18      if prog != null:
19        S$_d$ = S$_d$ $\cup$ {p | p is the head symbol of a clause in prog}
20        T$_d$ = T$_d$ $\cup$ {(E$^+$,E$^-$)}
21        P$_d$ = P$_d$ $\cup$ prog
22    return S$_d$, T$_d$, P$_d$
```

proof to Prolog. If this step fails, Metagol tries to unify the atom with the head of a metarule and tries to bind the variables in a metarule to symbols in S. Metagol saves the substitutions and tries to prove the body of the metarule recursively through meta-interpretation. After proving all atoms, Metagol induces a logic program by projecting the substitutions onto the corresponding metarules. Metagol checks the consistency of the induced program with the negative examples. If the program is inconsistent, Metagol backtracks to consider alternative programs. Metagol uses iterative deepening to ensure that the learned program has the minimal number of clauses. Metagol supports predicate invention by adding $d - 1$ new predicate symbols to S at each depth d.

4.2 Forgetgol

Forgetgol (Algorithm 1) takes as input (1) a logic program which represents BK, (2) a set of predicate symbols S which denotes those that may appear in a hypothesis, (3) a set of tasks T (a set of pairs of sets of positive and negative examples), (4) a maximum program size, and (5) a forgetting function. Forgetgol uses dependent learning (Lin et al. 2014) to learn programs and to expand B and S. Starting at depth $d = 1$, Forgetgol tries to learn a program for each task using at most d clauses. To learn a program, Forgetgol calls Metagol (line 17). Once Forgetgol has tried to learn a program for each task at depth d, it updates B (line 6) with the learned programs, increases the depth, and tries to learn programs for the remaining tasks (line 8). Forgetgol also updates S with learned predicate symbols (line 7), including invented symbols. Forgetgol repeats this process until it reaches a maximum depth, at which point it returns the learned programs. In contrast to the approach used by Lin et al, Forgetgol can also remove (forget) elements from S (line 4).

Algorithm 2 Syntactical forgetting

```
1   func forget(S,B):
2     S$'$ = {}
3     B$'$ = {}
4     for clause in B:
5       s = head_sym(clause)
6       if s in S:
7         clause$'$ = unfold(clause,B)
8         if clause$'$ is new with respect to B$'$:
9           S$'$ = S$'$ $\cup$ {s}
10          B$'$ = B$'$ $\cup$ {clause$'$}
11    return S$'$
```

4.3 Forgetting

The purpose of forgetting is to reduce the size of the hypothesis space without excluding the target hypothesis. Forgetgol reduces the size of the hypothesis space by reducing the number of predicate symbols allowed in a hypothesis (the set S in Algorithm 1). To be clear: Forgetgol does not remove clauses from the BK. Forgetgol removes predicate symbols from S because, as stated, the number of predicate symbols determines the size of the hypothesis space (Proposition 1). We consider two forgetting methods: *syntactical* and *statistical*.

Syntactical Forgetting Algorithm 2 shows our syntactical forgetting method, which removes syntactically duplicate programs. We use the Tamaki and Sato unfold operation (Tamaki and Sato 1984) on each clause in the BK to replace invented predicates with their definitions (but not when the predicate refers to the head of the clause, i.e. when it is used recursively). For each unfolded clause, we check whether (1) the head symbol of the clause is in S (i.e. is allowed in a hypothesis), and (2) we have already seen a clause with the same head arguments and the same body (line 8). If so, we forget the head predicate symbol; otherwise, we keep the symbol and add the unfolded clause to the seen set.

Statistical Forgetting Statistical forgetting is based on the hypothesis space results in Section 3. Deciding whether to keep or forget a predicate symbol (an element of S) in MIL depends on (1) the cost of relearning it, and (2) how likely it is to be reused. As Proposition 1 states, given p predicate symbols and m metarules with at most j body literals, the number of programs expressible with n clauses is at most $(mp^{j+1})^n$. Table 2 uses this result to show the costs of saving or forgetting a predicate symbol when it is relevant or irrelevant. However, we do not know beforehand whether a predicate symbol is relevant to future tasks. We can, however, estimate relevancy in our multi-task setting using the relative reuse of a symbol, similar to Dechter et al. (2013). Specifically, we define the probability $Pr(s, B)$ that a predicate symbol s is relevant given background knowledge B as:

$$Pr(s,B) = \frac{|\{clause \in B|\ \textit{s in body of clause}\ \}| + 1}{|B| + 1}$$

The +1 values denote additive smoothing. With this probability, we define the expected cost *cost_keep(s,B)* of keeping a

	Relevant	Irrelevant
Keep	$(m(p+1)^{j+1})^{n-k}$	$(m(p+1)^{j+1})^n$
Forget	$(mp^{j+1})^n$	$(mp^{j+1})^n$

Table 2: The costs of keeping or forgetting a predicate symbol that is the head symbol of a definition composed of k clauses. These costs are based on Proposition 1, where p is the number of predicate symbols, m is the number of metarules with at most j body literals, and n is the number of clauses in the target program. The $n-k$ exponent when keeping a relevant symbol is because reusing a learned symbol reduces the size of the target program by k clauses.

Algorithm 3 Statistical forgetting

```
func forget(S,B):
  $S' = \{\}$
  for clause in B:
    s = head_sym(clause)
    if s in S and cost_forget(s,B) $>$ cost_keep(s,B):
      $S' = S' \cup \{s\}$
  return $S'$
```

symbol s that is the head of k clauses when searching for a program with n clauses:

$$\textit{cost\_keep(s,B)} = \begin{array}{l}(pr(s,B)(m(p+1)^{j+1})^{n-k}) \\ +\,((1-pr(s,B))(m(p+1)^{j+1})^n)\end{array}$$

We can likewise define the expected cost *cost_forget(s,B)* = $(mp^{j+1})^n$ of forgetting s. These costs allow us to define the forget function shown in Algorithm 3.

5 Experiments

To test our claim that forgetting can improve learning performance, our experiments aim to answer the question:

Q1 Can forgetting improve learning performance?

To answer **Q1** we compare three variations of Forgetgol:

- **Forgetgol$_{\text{syn}}$**: Forgetgol with syntactical forgetting
- **Forgetgol$_{\text{stat}}$**: Forgetgol with statistical forgetting
- **Metabias**: Forgetgol set to remember everything (we call Algorithm 1 with a forget function that returns all the BK), which is equivalent to approach used in (Lin et al. 2014)

We introduced forgetting because we claim that learners which remember everything will become overwhelmed by too much BK when learning from many tasks. To test this claim, our experiments aim to answer the question:

Q2 Do remember everything learners become overwhelmed by too much BK when learning from many tasks?

To answer **Q2**, we compare the performance of the learners on progressively more tasks. We expect that Metabias will improve given more tasks but will eventually become overwhelmed by too much BK, at which point its performance will start to degrade. By contrast, we expect that Forgetgol will improve given more tasks and should outperform Metabias when given many tasks because it can forget BK.

We also compare Forgetgol and Metabias against Metagol. However, this comparison cannot help us answer questions **Q1** and **Q2**, which is also why we do not compare Forgetgol against other program induction systems. The purpose of this comparison is to add more experimental evidence to the results of Lin et al. (2014), where they compare multi-task learning with single-task learning. We expect that Metagol will not improve given more tasks because it cannot reuse learned programs.

All the experimental data are available at https://github.com/andrewcropper/aaai20-forgetgol.

5.1 Experiment 1 - Robot Planning

Our first experiment is on learning robot plans.

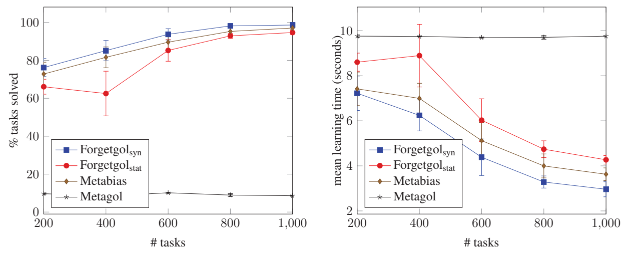
Materials A robot and a ball are in a 6 × 6 space. A state describes the position of the robot, the ball, and whether the robot is holding the ball. A training example is an atom $f(s_1, s_2)$, where f is the target predicate and s_1 and s_2 are initial and final states respectively. The task is to learn a program to move from the initial to the final state. We generate training examples by generating random states, with the constraint that the robot can only hold the ball if it is in the same position as the ball. The robot can perform the actions *up*, *down*, *right*, *left*, *grab*, and *drop*. The learners use the metarules in Table 1.

Method Our dataset contains n tasks for each n in $\{2000, 4000, \ldots, 20000\}$. Each task is a one-shot learning task and is given a unique predicate symbol. We enforce a timeout of 60 seconds per task per search depth. We set the maximum program size to 6 clauses. We measure the percentage of tasks solved (tasks where the learner learned a program) and learning times. We plot the standard error of the means over 5 repetitions.

Results Figure 1 shows the results on the small dataset. As expected, Metagol's performance does not vary when given more tasks. By contrast, Forgetgol and Metabias continue to improve given more tasks, both in terms of percentage of tasks solved and learning time. Figure 2 shows the results on the big dataset. We did not run Metagol on the big dataset because the results on the small dataset are sufficiently conclusive. On the big dataset, the performances of Metabias and Forgetgol$_{\text{stat}}$ start to degrade when given more than 8000 tasks. However, this performance decrease is tiny ($<$1%). By contrast, Forgetgol$_{\text{syn}}$ always solves 100% of the tasks and by 20,000 tasks learns programs twice as quick as Metabias and Forgetgol$_{\text{stat}}$.

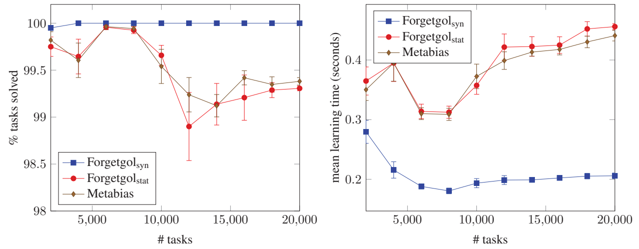
These results are somewhat unexpected. We thought that Metabias would eventually become overwhelmed by too much BK, at which point its performance would start to degrade. Metabias does not strongly exhibit this behaviour (the performance decrease is tiny) and performs well despite large amounts of BK. After analysing the experimental results, Metabias rarely has to learn a program with more than two clauses because it could almost always reuse a learned program. The performance of Forgetgol$_{\text{stat}}$ is similar to Metabias because Forgetgol$_{\text{stat}}$ rarely forgets anything, which suggests limitations with the proposed statistical forgetting method.

Overall, these results suggest that the answer to **Q1** is yes and the answer to **Q2** is no.



(a) Percentage of tasks solved (b) Mean learning time

Figure 1: Robot experiment small dataset results.



(a) Percentage of tasks solved (b) Mean learning time

Figure 2: Robot experiment big dataset results.

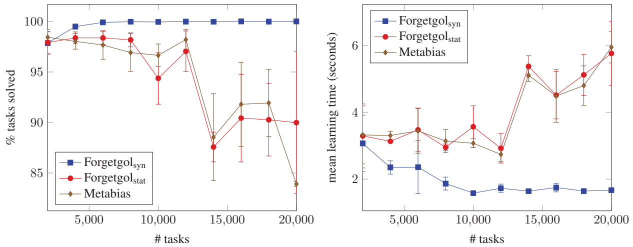
5.2 Experiment 2 - Lego

Our second experiment is on building Lego structures.

Materials We consider a Lego world with dimensionality 6×1. For simplicity we only consider 1×1 blocks of a single colour. A training example is an atom $f(s_1, s_2)$, where f is the target predicate and s_1 and s_2 are initial and final states respectively. A state describes the Lego board as a list of integers. The value k at index i denotes that there are k blocks stacked at position i. The goal is to learn a program to build the Lego structure from a blank Lego board (a list of zeros). We generate training examples by generating random final states. The learner can move along the board using the actions *left* and *right*; can place a Lego block using the action *place_block*; and can use the fluents *at_left* and *at_right* and their negations to determine whether it is at the leftmost or rightmost board position. The learners use the metarules in Table 1.

Method The method is the same as in experiment 1, except we only use a big dataset.

Results The results (Figure 3) show that the performance of Metabias substantially worsens after 12,000 tasks. By 20,000 tasks Metabias can only solve around 80% of the tasks. By contrast, Forgetgol$_{syn}$ always solves 100% of the tasks and by 20,000 tasks can learn programs three times quicker than Metabias. The performance of Forgetgol$_{stat}$ is again similar to Metabias. These results strongly suggest that the answers to **Q1** and **Q2** are both yes.



(a) Percentage of tasks solved (b) Mean learning time

Figure 3: Lego experiment results.

6 Conclusions and Limitations

We have explored the idea of *forgetting*. In this approach, a program induction system learns programs over time and is able to revise its BK (and thus its hypothesis space) by adding and removing (forgetting) learned programs. Our theoretical results show that forgetting can substantially reduce the size of the hypothesis space (Theorem 2) and the sample complexity (Theorem 3) of a learner . We implemented our idea in Forgetgol, a new ILP learner. We described two forgetting techniques: syntactical and statistical. Our experimental results on two domains show that (1) forgetting can substantially improve learning performance (answers yes to **Q1**), and (2) remember everything approaches become overwhelmed by too much BK (answers yes to **Q2**).

6.1 Limitations and Future Work

There are many limitations to this work, including (1) only considering two domains, (2) only using Forgetgol with Metagol, and (3) not evaluating the impact of forgetting on predictive accuracy. However, future work can easily address these minor limitations.

The main limitation of this work, and thus the main topic for future research, is our forgetting methods. Syntactical forgetting achieves the best performance, but if every learned program is syntactically unique, then this method will forget nothing. Future work can address this limitation by exploring semantic forgetting methods, such as those based on subsumption (Plotkin 1971) or derivability (Cropper and Tourret 2019). Statistical forgetting, based on expected search cost, does not perform better than Metabias (which remembers everything). Statistical forgetting rarely forgets programs because the expected search cost is dominated by the target program size. Future work could improve this method by adding a *frugal* dampening factor to force Forgetgol to forget more. We suspect that another limitation with our forgetting methods is that they will be sensitive to concept drift (Widmer and Kubat 1996). For instance, if we first trained Forgetgol on the robot domain and then on the Lego domain, we would want it to revise its BK accordingly. Future work could address this limitation by adding an exponential decay factor to forget programs that have not recently been used.

To conclude, we think that forgetting is an important contribution to lifelong machine learning and that our work will stimulate future research into better forgetting methods.

References

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. Deepcoder: Learning to write programs. In *ICLR*. OpenReview.net.

Baxter, J. 2000. A model of inductive bias learning. *J. Artif. Intell. Res.* 12:149–198.

Beierle, C., and Timm, I. J. 2019. Intentional forgetting: An emerging field in AI and beyond. *KI* 33(1):5–8.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's razor. *Inf. Process. Lett.* 24(6):377–380.

Caruana, R. 1997. Multitask learning. *Machine Learning* 28(1):41–75.

Cropper, A., and Muggleton, S. H. 2016. Metagol system. https://github.com/metagol/metagol.

Cropper, A., and Muggleton, S. H. 2019. Learning efficient logic programs. *Machine Learning* 108(7):1063–1083.

Cropper, A., and Tourret, S. 2019. Logical reduction of metarules. *Machine Learning*. To appear.

Cropper, A.; Morel, R.; and Muggleton, S. H. 2019. Learning higher-order logic programs. *Machine Learning*. To appear.

Cropper, A. 2019. Playgol: Learning programs through play. In Kraus, S., ed., *IJCAI 2019*, 6074–6080. ijcai.org.

Dechter, E.; Malmaud, J.; Adams, R. P.; and Tenenbaum, J. B. 2013. Bootstrap learning via modular concept discovery. In *IJCAI 2013*, 1302–1309. IJCAI/AAAI.

Dietterich, T. G.; Domingos, P. M.; Getoor, L.; Muggleton, S.; and Tadepalli, P. 2008. Structured machine learning: the next ten years. *Machine Learning* 73(1):3–23.

Dumancic, S., and Blockeel, H. 2017. Clustering-based relational unsupervised representation learning with an explicit distributed representation. In Sierra, C., ed., *IJCAI 2017*, 1631–1637. ijcai.org.

Dumancic, S.; Guns, T.; Meert, W.; and Blockeel, H. 2019. Learning relational representations with auto-encoding logic programs. In Kraus, S., ed., *IJCAI 2019*, 6081–6087. ijcai.org.

Ellis, K.; Morales, L.; Sablé-Meyer, M.; Solar-Lezama, A.; and Tenenbaum, J. 2018a. Learning libraries of subroutines for neurally-guided bayesian program induction. In *NeurIPS 2018*, 7816–7826.

Ellis, K.; Ritchie, D.; Solar-Lezama, A.; and Tenenbaum, J. 2018b. Learning to infer graphics programs from hand-drawn images. In *NeurIPS 2018*, 6062–6071.

Evans, R., and Grefenstette, E. 2018. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.* 61:1–64.

Ferri, C.; Hernández-Orallo, J.; and Ramírez-Quintana, M. J. 2001. Incremental learning of functional logic programs. In Kuchen, H., and Ueda, K., eds., *FLOPS 2001*, volume 2024 of *Lecture Notes in Computer Science*, 233–247. Springer.

Heule, M.; Järvisalo, M.; Lonsing, F.; Seidl, M.; and Biere, A. 2015. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.* 53:127–168.

Lake, B. M.; Ullman, T. D.; Tenenbaum, J. B.; and Gershman, S. J. 2017. Building machines that learn and think like people. *Behavioral and brain sciences* 40.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive learning of answer set programs. In *JELIA 2014*, 311–325.

Lin, F., and Reiter, R. 1994. Forget it. In *Working Notes of AAAI Fall Symposium on Relevance*, 154–159.

Lin, D.; Dechter, E.; Ellis, K.; Tenenbaum, J. B.; and Muggleton, S. 2014. Bias reformulation for one-shot function induction. In *ECAI 2014*, 525–530.

Lloyd, J. W. 2012. *Foundations of logic programming*. Springer Science & Business Media.

Maloof, M. A. 1997. *Progressive Partial Memory Learning*. Ph.D. Dissertation, Fairfax, VA, USA. UMI Order No. GAX97-10122.

Martínez-Plumed, F.; Ferri, C.; Hernández-Orallo, J.; and Ramírez, M. J. 2015. Knowledge acquisition with forgetting: an incremental and developmental setting. *Adaptive Behaviour* 23(5):283–299.

McCloskey, M., and Cohen, N. J. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24. Elsevier. 109–165.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning* 100(1):49–73.

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

Plotkin, G. 1971. *Automatic Methods of Inductive Inference*. Ph.D. Dissertation, Edinburgh University.

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

Raedt, L. D. 2008. *Logical and relational learning*. Cognitive Technologies. Springer.

Sablon, G., and Raedt, L. D. 1995. Forgetting and compacting data in concept learning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, 432–438. Morgan Kaufmann.

Silver, D. L.; Yang, Q.; and Li, L. 2013. Lifelong machine learning systems: Beyond learning algorithms. In *Lifelong Machine Learning, Papers from the 2013 AAAI Spring Symposium, Palo Alto, California, USA, March 25-27, 2013*, volume SS-13-05 of *AAAI Technical Report*. AAAI.

Srinivasan, A.; King, R. D.; and Bain, M. 2003. An empirical study of the use of relevance information in inductive logic programming. *J. Mach. Learn. Res.* 4:369–383.

Srinivasan, A.; Muggleton, S. H.; and King, R. D. 1995. Comparing the use of background knowledge by inductive logic programming systems. In *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 199–230. Department of Computer Science, Katholieke Universiteit Leuven.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*.

Stahl, I. 1995. The appropriateness of predicate invention as bias shift operation in ILP. *Machine Learning* 20(1-2):95–117.

Tamaki, H., and Sato, T. 1984. Unfold/fold transformation of logic programs. In Tärnlund, S., ed., *Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, July 2-6, 1984*, 127–138. Uppsala University.

Torrey, L., and Shavlik, J. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI Global. 242–264.

Valiant, L. G. 1984. A theory of the learnable. *Commun. ACM* 27(11):1134–1142.

Widmer, G., and Kubat, M. 1996. Learning in the presence of concept drift and hidden contexts. *Machine Learning* 23(1):69–101.