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 experimentally compare forgetting against approaches that either remember or forget everything. Our experimental results show that forgetting 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 choose 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) 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.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
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 learning (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 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 rewrites 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. Deeppcoder (Balog et al. 2017) and Dreamcoder (Ellis et al. 2018a) both train neural networks to score how relevant programs are in getting is essentially identifying irrelevant BK. 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 $\mathcal{V}_{B,E^+,E^-} = \{ H \in \mathcal{H}_B | (H \cup B \models E^+) \land (\forall e^- \in E^-, H \cup B \not\models e^-) \}$.

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 $B' \subseteq B$ 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 \( H \in \text{Hypothesis space} \) and there is no \( B'' \subset B' \) such that \( H \in \text{Hypothesis space} \). 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. \( \square \)

### 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 \textit{grandparent} relation, background knowledge with the \textit{parent} relation, and the metarules in Table 1, a MIL learner could use the \textit{chain} metarule with the substitutions \{P/\textit{grandparent}, Q/\textit{parent}, R/\textit{parent}\} to induce the theory: \textit{grandparent}(A,B) \leftarrow \textit{parent}(A,C), \textit{parent}(C,B).

<table>
<thead>
<tr>
<th>Name</th>
<th>Metarule</th>
</tr>
</thead>
<tbody>
<tr>
<td>ident</td>
<td>( P(A, B) \leftarrow Q(A, B) )</td>
</tr>
<tr>
<td>precon</td>
<td>( P(A, B) \leftarrow Q(A), R(A, B) )</td>
</tr>
<tr>
<td>postcon</td>
<td>( P(A, B) \leftarrow Q(A, B), R(B) )</td>
</tr>
<tr>
<td>chain</td>
<td>( P(A, B) \leftarrow Q(A, C), R(C, B) )</td>
</tr>
</tbody>
</table>

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 \( \text{sig}(P) \) 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 \( p = |\text{sig}(B)| \), \( m \) 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 \( \epsilon \) and confidence \( \delta \), 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, \( p = |\text{sig}(B)|, p' = |\text{sig}(B')|, n \) 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 \textit{quicksort} which uses \textit{partition} and \textit{append} but we may want Metagol to only use \textit{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

```plaintext
func forgetgol(B,S,T,max_d,forget):
  P = {};
  for depth=1 to max_d:
    S' = forget(S,B);
    S_d, T_d, P_d = learn(B,S',T,depth)
    B = B ∪ P_d
    S = S ∪ S_d
    T = T \ T_d
    P = P ∪ P_d
  return P
```

Algorithm 2 Syntactical forgetting

```plaintext
func forget(S,B):
  S' = {};
  for clause in B:
    clause' = unfold(clause,B)
    if clause' is new with respect to B':
      S' = S' ∪ {s}
      B' = B' ∪ {clause'}
  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:

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

The +1 values denote additive smoothing. With this probability, we define the expected cost $cost_{keep}(s,B)$ of keeping a
We introduced forgetting because we claim that learners which remember everything will become overwhelmed by too much BK, at which point its performance would 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 $6 \times 6$ 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, ..., 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 stat start to degrade when given more than 8000 tasks. However, this performance decrease is tiny (<1%). By contrast, Forgetgol syn always solves 100% of the tasks and by 20,000 tasks learns programs twice as quick as Metabias and Forgetgol 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 stat is similar to Metabias because Forgetgol stat rarely forgets anything, which suggests limitations with the proposed statistical forgetting method.

<table>
<thead>
<tr>
<th>Keep</th>
<th>Relevant</th>
<th>$\frac{(m(p+1)^{j+1})^{n-k}}{(mp^{j+1})^n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forget</td>
<td>Irrelevant</td>
<td>$\frac{(m(p+1)^{j+1})^n}{(mp^{j+1})^n}$</td>
</tr>
</tbody>
</table>

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

1 function forget (S,B):
2    S' = {}
3    for clause in B:
4        s = head_sym(clause)
5        if s in S and cost_forget (s,B) > cost_keep(s,B):
6            S' = S' ∪ {s}
7    return S'
```

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

```
cost_keep(s,B) = \left( p r(s,B)(m(p + 1)^{j+1})^{n-k} + (1 - p r(s,B))(m(p + 1)^{j+1})^n \right)
```

We can likewise define the expected cost $\text{cost_forget}(s, B) = (mp^{j+1})^n$ of forgetting $s$. These costs allow us to define the forget function shown in Algorithm 3.
Overall, these results suggest that the answer to Q1 is yes and the answer to Q2 is no.

Figure 1: Robot experiment small dataset results.

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 \times 1. For simplicity we only consider 1 \times 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.
References


