Learning Product Automata

Revisiting a trick by Rivest and Schapire

Joshua Moerman
Institute for Computing and Information Sciences
Radboud University, Nijmegen, the Netherlands
joshua.moerman@cs.ru.nl

Abstract—In this paper we give an optimization for active learning algorithms, applicable to learning Moore machines where the output comprises several observables. These machines can be decomposed themselves by projecting on each observable, resulting in smaller components. These components can then be learnt with fewer queries. This is in particular interesting for learning software, where compositional methods are important for guaranteeing scalability.

I. INTRODUCTION

Active automata learning is becoming a valuable tool in software engineering and verification [1]. Indeed, applications can be found in a broad range: finding bugs in network protocols [2], assisting with refactoring legacy software [3], specification mining [4], and more.

These learning techniques originate from the field of grammatical inference. One of the crucial steps for applying these techniques on software was to move from deterministic finite automata to deterministic Moore or Mealy machines, capturing reactive systems with any kind of output. With little adaptations, the algorithms work well, as shown by the many applications. This is remarkable, since little software specific knowledge is used (besides the input alphabet of actions).

Realizing that software is often composed of smaller pieces, it makes sense to incorporate such information in learning algorithms. In the present paper we aim to do exactly that for the simplest case of composition: we learn product automata.

To the best of the author’s knowledge, this has not been done before explicitly. Furthermore, libraries such as LearnLib [5] and libalf [6] do not include such functionality. Implicitly, however, it has been done before. Rivest and Schapire use two tricks to reduce the size of some automata in their paper “Diversity-based inference of finite automata” [7]. The first trick is to look at the reversed automaton (in their terminology, the diversity-based automaton). The second trick (which is not explicitly mentioned, unfortunately) is to have a different automaton for each observable (i.e. output). In one of their examples the two tricks combined give a reduction from $\pm10^{19}$ states to just 54 states.

We isolate this trick, so we can apply it more generally. Furthermore, we argue that this is particularly interesting in the context of model learning of software, as composition is a common tool in software engineering.

II. PRELIMINARIES

We use the formalism of Moore machines to describe our algorithms. Nonetheless, the results can also be phrased in terms of Mealy machines.

Definition 1: A Moore machine is a tuple $M = (Q, I, O, \delta, o, q_0)$ where $Q, I$ and $O$ are finite sets of states, inputs and outputs respectively, $\delta : Q \times I \rightarrow Q$ is the transition function, $o : Q \rightarrow O$ is the output function, and $q_0$ is the initial state. The size $|M|$ is the cardinality of $Q$.

We extend the definition of the transition function to words as $\delta : Q \times I^* \rightarrow Q$. The behaviour of a state $q$ is the map $[q] : I^* \rightarrow O$ defined by $[q](w) = o(\delta(q, w))$. We extend this to the machine $[M] = [q_0]$. Two states $q, q'$ are equivalent if $[q] = [q']$. Two machines are equivalent if their initial states are. A machine is minimal if all states have different behaviour and all states are reachable.

Definition 2: Given two Moore machines with equal input sets $M_1 = (Q_1, I_1, Q_1, \delta_1, o_1, q_{01})$ and $M_2 = (Q_2, I_2, Q_2, \delta_2, o_2, q_{02})$, we define their product $M_1 \times M_2$ by:

$$M_1 \times M_2 = (Q_1 \times Q_2, I_1 \times O_2, \delta_1 \times \delta_2, (q_{01}, q_{02})),$$

where $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ and $o((q_1, q_2)) = (o_1(q_1), o_2(q_2))$.

The product is formed by running both machines in parallel and letting $I$ act on both machine simultaneously. The output of both machines is observed. Note that the product Moore machine might have unreachable states, even if the components are reachable. The product of more than two machines is defined by induction.

Let $M$ be a machine with outputs in $O_1 \times O_2$. By post-composing the output function with projection functions we get two machines, called components, $M_1$ and $M_2$ with outputs in $O_1$ and $O_2$ respectively. Then $M$ is equivalent to $M_1 \times M_2$. If $M$ and its components $M_i$ are taken to be minimal, then we have $|M| \leq |M_1| \cdot |M_2|$ and $|M_i| \leq |M|$. In the best case we have $|M_i| = \sqrt{|M|}$ and so the behaviour of $M$ can be described using only $2\sqrt{|M|}$ states, which is less than $|M|$ (if $|M| > 4$). With iterated products the reduction can be even more as shown in the following example. This reduction in state-space can be exploited by learning algorithms, as will be shown in later sections.

We introduce basic notation: $\pi_i : A_1 \times A_2 \rightarrow A_i$ are the usual projection functions. On a function $f : X \rightarrow A_1 \times A_2$ we
Fig. 1. A state of the 8-bit machine

use the shorthand \( \pi_i f \) to denote \( \pi_i \circ f \). As usual, \( uv \) denotes concatenation of string \( u \) and \( v \), and this is lifted to sets of strings \( UV = \{ uv \mid u \in U, v \in V \} \). We define the set \([n] = \{1, \ldots, n\}\) and the set of Boolean values \( \mathbb{B} = \{0, 1\} \).

A. Example

We take the \( n \)-bit register machine example from [7]. The state space of the \( n \)-bit register machine \( M_n \) is given by \( n \) bits and a position of the reading/writing head, see Figure 1. The inputs are commands to control the position of the head and to flip the current bit. The output is the current bit vector. Formally it is defined as \( M_n = (\mathbb{B}^n \times [n], \{L,R,F\}, \mathbb{B}^n, \delta, o, i) \), where the initial state is \( i = ((0, \ldots, 0), 1) \). The output is \( o(((b_1, \ldots, b_n), k)) = (b_1, \ldots, b_n) \) and the transition function is given by

\[
\delta(((b_1, \ldots, b_n), k), L) = \begin{cases} 
(b_1, \ldots, b_n), k-1 \quad & \text{if } k > 1 \\
(b_1, \ldots, b_n), n \quad & \text{if } k = 1 
\end{cases}
\]

\[
\delta(((b_1, \ldots, b_n), k), R) = \begin{cases} 
(b_1, \ldots, b_n), k+1 \quad & \text{if } k < n \\
(b_1, \ldots, b_{n-1}, 1) \quad & \text{if } k = n 
\end{cases}
\]

\[
\delta(((b_1, \ldots, b_n), k), F) = \begin{cases} 
(b_1, \ldots, b_{n-1}, b_n), k \quad & \text{if } L \text{ moves the head to the left and } R \text{ to the right (and wraps around on the ends), while } F \text{ flips the current bit.}
\end{cases}
\]

The machine \( M_n \) is minimal and has \( n \cdot 2^n \) states. So although this machine has very simple behaviour, learning it will require a lot of queries because of its size. Luckily, the machine can be decomposed into smaller components. For each bit \( l \) we define a component \( M^l_n = (\mathbb{B} \times [n], \{L,R,F\}, \mathbb{B}, \delta^l, o^l, (0,1)) \) where \( o^l((b,k)) = b \) and

\[
\delta^l((b,k), L) = \begin{cases} 
(b,k-1) \quad & \text{if } k > 1 \\
(b,n) \quad & \text{if } k = 1
\end{cases}
\]

\[
\delta^l((b,k), R) = \begin{cases} 
(b,k+1) \quad & \text{if } k < n \\
(b,1) \quad & \text{if } k = n
\end{cases}
\]

\[
\delta^l((b,k), F) = \begin{cases} 
(-b,k) \quad & \text{if } l = k \\
(b,k) \quad & \text{if } l \neq k
\end{cases}
\]

The product \( M^1_n \times \cdots \times M^n_n \) is equivalent to \( M_n \). Each of the components \( M^l_n \) is minimal and has only \( 2n \) states. So by this decomposition, we only need \( 2 \cdot n^2 \) states to describe the whole behaviour of \( M_n \). Note, however, that the product \( M^1_n \times \cdots \times M^n_n \) is not minimal: many states are unreachable.

III. Learning

We describe two approaches for active learning of product machines. One is a direct extension of the well-known \( \text{L}^* \) algorithm. The other reduces the problem to any active learning algorithm, so that one can use more optimised algorithms.

We fix an unknown target machine \( M \) with a known input alphabet \( I \) and output alphabet \( O = O_1 \times O_2 \). The goal of the learning algorithm is to infer a machine equivalent to \( M \), given access to a \textit{minimally adequate teacher} [8]. The teacher will answer two types of queries:

- \textit{Membership queries} \( MQ(w) \) for words \( w \in I^* \), the teacher will answer with \( \{M\}(w) \in O \).
- \textit{Equivalence queries} \( EQ(M') \) for a machine \( M' \) on the same alphabets, the teacher will answer with \( \text{YES} \) if \( M \) and \( M' \) are equivalent and she will answer with a word \( w \) such that \( \{M\}(w) \neq \{M'\}(w) \) otherwise.

A. Learning product automata with an \( \text{L}^* \) extension

We can use the general framework for automata learning as set up in [9], [10]. The general account does not directly give concrete algorithms, but it does give generalised definitions for \textit{closedness} and \textit{consistency}. The main data structure for the algorithm is an observation table.

Definition 3: An observation table is a triple \( (S,E,T) \) where \( S,E \subseteq I^* \) are finite sets of words and \( T : S \cup E \rightarrow O^E \) is defined by \( T(s)(e) = \{M\}(se) \).

During the \( \text{L}^* \) algorithm the sets \( S,E \) grow and \( T \) encodes the knowledge of \( \{M\} \) so far.

Definition 4: An observation table \( (S,E,T) \) is product-closed \( \star \) if for all \( t \in SI \) there exist \( s_1,s_2 \in S \) such that \( \pi_i T(t) = \pi_i T(s_i) \) for \( i = 1, 2 \). It is product-consistent if for \( i = 1,2 \) and for all \( s,s' \in S \) we have \( \pi_i T(s) = \pi_i T(s') \) implies \( \pi_i T(sa) = \pi_i T(s'a) \) for all \( a \in I \).

These definitions are related to the classical definitions of closedness and consistency. In fact the latter two points of the following lemma restate the above definitions. For the first two the converse does not necessarily hold.

Lemma 1: Let \( OT = (S,E,T) \) be an observation table and let \( \pi_i OT = (S,E,\pi_i T) \) be a component. We have the following implications:

1) \( OT \) is closed \( \implies \) \( OT \) is product-closed,
2) \( OT \) is consistent \( \iff \) \( OT \) is product-consistent,
3) \( OT \) is product-closed \( \iff \) \( \forall i \), \( \pi_i OT \) is closed \( \forall i \),
4) \( OT \) is product-consistent \( \iff \) \( \pi_i OT \) is consistent \( \forall i \).

Lemma 2: Given a product-closed and -consistent table we can define a product Moore machine consistent with the table, where each component is minimal.

We list the product-\( \text{L}^* \) algorithm in Figure 2. Its termination follows from the fact that \( \text{L}^* \) terminates on both components.

By Lemma [1](1) we note that the algorithm does not need more rows than we would need by running \( \text{L}^* \) on \( M \). By point (4) of the same lemma, we find that it does not need more columns than \( \text{L}^* \) would need on each component combined. This means that in the worst case, the table is twice as big as the original \( \text{L}^* \) would do. However, in good cases (such as the running example), the table is much smaller, as the number of rows is less for each component and the columns needed for each component may be similar.
B. Learning product automata via a reduction

The previous algorithm constructs two machines from a single table. This suggests that we can also run two learning algorithms to construct two machines. We lose the fact that the data structure is shared between the learners, but we gain that we can use more efficient algorithms than L* without any effort.

The crucial observation is that a counterexample is necessarily a counterexample for at least one of the two learners. In this case we simply forward the counterexample to that learner. (If at a certain stage only one learner makes an error, we keep the other learner suspended, as we may obtain a counterexample for that one later on.) This observation means that at least one of the learners makes progress and will eventually terminate. Hence, the whole algorithm will terminate.

In the worst case, twice as many queries will be posed, compared to learning the whole machine at once. (This is because learning the full machine also learns its components.) In good cases, such as the running example, it requires much less queries. Typical learning algorithms require $O(n^2)$ membership queries ($n$ being the number of states of the minimal machine). For the example $M_n$ this gives $O((n \cdot 2^n)^2) = O(n^2 \cdot 2^{2n})$ queries. When learning the components $M'_n$ with the above algorithm, that gives just $O((2n)^2 + \cdots + (2n)^2) = O(n^3)$ queries.

IV. Experiments

The algorithm via reduction is implemented in LearnLib. As we expect the reduction algorithm to be the most efficient (as it can use an efficient learner internally), we leave an implementation of the direct extension of $L^*$ as future work.

---

1. The implementation and models can be found on-line at [https://gitlab.science.ru.nl/noorman/learning-product-automata](https://gitlab.science.ru.nl/noorman/learning-product-automata)

2. The original files describing these circuits can be found at [https://people.engr.ncsu.edu/brglez/CBL/benchmarks/](https://people.engr.ncsu.edu/brglez/CBL/benchmarks/)
58 128 states, much bigger than the target machine of 202 states. In the theoretical framework, this is not an issue, as the teacher will allow it and answer the query. Even in the PAC model, this poses no problem as we can efficiently determine membership. However, in some applications the equivalence queries are implemented with a model checker or some sophisticated test generation tool \[2\]. In these cases, the increased hypotheses may be undesirable.

V. Final Remarks

We have shown two learning algorithms which exploit a decomposable output. If the output can be split, then also the machine itself can be decomposed in components. As the few experiments show, this can be a very effective optimization for learning software. It should be stressed that the improvement of the optimization depends heavily on the independence of the components. For example, the \(n\)-bit register machine has nearly independent components and the reduction in the number of queries is big. The more realistic circuits did not show such improvements. A potential problem is the growth of the intermediate hypotheses. In the remainder of this section we discuss related ideas left for future work.

A. Generalization to subsets of products

In some cases we might know even more about our output alphabet. The output set \(O\) may be a proper subset of \(O_1 \times O_2\), indicating that some outputs can only occur "synchronised". For example, we might have \(O = \{(0, 0)\} \cup \{(a, b) | a, b \in [3]\}\), i.e. the output 0 for either component can only occur if the other component is also 0.

In such cases we can use the above algorithm still, but we may insist that the teacher only accepts machines with output in \(O\) for the equivalence queries (as opposed to outputs in \(\{0, 1, 2, 3\}\)). When constructing \(H = H_1 \times H_2\) in line 7 of Figure 3 we can do a reachability analysis on \(H\) to check for non-allowed outputs. If such traces exist, we know it is a counterexample for at least one of the two learners. With such traces we can fix the defect ourselves, without having to rely on the teacher.

B. The other trick by Rivest and Schapire

The main trick of \[7\] was to exploit the structure of the so-called "diversity-based" automaton. This automaton may also be called the reversed Moore machine. It provides a duality between reachability and equivalence which is the core idea of Brzozowski's minimization algorithm \[13, 14\].

Let \(M^R\) denote the reverse of \(M\), then we have \([M^R](w) = [M](w^R)\). This allows us to give an \(L^*\) algorithm which learns \(M^R\) by posing membership queries with the words reversed. We computed \(M^R\) for the circuit models and all but one of them was much larger than the original. This suggests that it might not be useful as a trick in learning software, however, a more thorough investigation is desired.

C. Other types of composition

In this paper, we only looked at the simplest type of composition: products of outputs. On the other side we can look at combining inputs. By taking the disjoint union of inputs sets \(I_1\) and \(I_2\) we can run two machines in parallel, the first is acted upon by \(I_1\) and the second by \(I_2\). In this simple construction the machines are fully parallel. That is, the inputs from \(I_1\) commute with the inputs from \(I_2\). More generally, it is interesting to investigate what happens if there are interactions.

A more complex type of product is the cascaded product (also called the wreath product) where the transition structure of the second component may depend on the output (or even state) of the first component. This captures hierarchical composition \[15\]. The connection of these more general compositions and learning is left as future work.

### Table I

| Machine | States | Components | Product learner | TTT learner |
|---------|--------|------------|----------------|-------------|
|         |        | EQs | MQs | Actions | EQs | MQs | Actions |
| \(M_2\) | 8      | 2  | 3  | 100   | 621 | 5   | 115  | 869 |
| \(M_3\) | 24     | 3  | 3  | 252   | 1555| 5   | 347  | 2946|
| \(M_4\) | 64     | 4  | 8  | 456   | 3025| 6   | 1058 | 13824|
| \(M_5\) | 160    | 5  | 6  | 809   | 7665| 17  | 2723 | 34057|
| \(M_6\) | 384    | 6  | 11 | 1383  | 12870| 25  | 6250 | 90370|
| \(M_7\) | 896    | 7  | 11 | 2087  | 24156| 52  | 14627 | 226114|
| \(M_8\) | 2048   | 8  | 13 | 3289  | 41732| 100 | 34042 | 651678|
| bbara   | 7      | 2  | 3  | 167   | 1049| 3   | 216  | 1535|
| mark1   | 202    | 8^*| 22 | 13027 | 117735| 67  | 15192 | 252874|
| keyb    | 41     | 2  | 25 | 12464 | 153899| 24  | 6024 | 265805|
| ex3     | 28     | 2  | 24 | 1133  | 9042 | 18  | 878  | 91494|

Fig. 4. The number of states for each hypothesis while learning mark1.

In some cases we might know even more about our output alphabet. The output set \(O\) may be a proper subset of \(O_1 \times O_2\), indicating that some outputs can only occur “synchronised”.

For example, we might have \(O = \{(0, 0)\} \cup \{(a, b) | a, b \in [3]\}\), i.e. the output 0 for either component can only occur if the other component is also 0.

In such cases we can use the above algorithm still, but we may insist that the teacher only accepts machines with output in \(O\) for the equivalence queries (as opposed to outputs in \(\{0, 1, 2, 3\}\)). When constructing \(H = H_1 \times H_2\) in line 7 of Figure 3 we can do a reachability analysis on \(H\) to check for non-allowed outputs. If such traces exist, we know it is a counterexample for at least one of the two learners. With such traces we can fix the defect ourselves, without having to rely on the teacher.

B. The other trick by Rivest and Schapire

The main trick of \[7\] was to exploit the structure of the so-called “diversity-based” automaton. This automaton may also be called the reversed Moore machine. It provides a duality between reachability and equivalence which is the core idea of Brzozowski’s minimization algorithm \[13, 14\].

Let \(M^R\) denote the reverse of \(M\), then we have \([M^R](w) = [M](w^R)\). This allows us to give an \(L^*\) algorithm which learns \(M^R\) by posing membership queries with the words reversed. We computed \(M^R\) for the circuit models and all but one of them was much larger than the original. This suggests that it might not be useful as a trick in learning software, however, a more thorough investigation is desired.

C. Other types of composition

In this paper, we only looked at the simplest type of composition: products of outputs. On the other side we can look at combining inputs. By taking the disjoint union of inputs sets \(I_1\) and \(I_2\) we can run two machines in parallel, the first is acted upon by \(I_1\) and the second by \(I_2\). In this simple construction the machines are fully parallel. That is, the inputs from \(I_1\) commute with the inputs from \(I_2\). More generally, it is interesting to investigate what happens if there are interactions.

A more complex type of product is the cascaded product (also called the wreath product) where the transition structure of the second component may depend on the output (or even state) of the first component. This captures hierarchical composition \[15\]. The connection of these more general compositions and learning is left as future work.

### Table I

| Machine | States | Components | Product learner | TTT learner |
|---------|--------|------------|----------------|-------------|
|         |        | EQs | MQs | Actions | EQs | MQs | Actions |
| \(M_2\) | 8      | 2  | 3  | 100   | 621 | 5   | 115  | 869 |
| \(M_3\) | 24     | 3  | 3  | 252   | 1555| 5   | 347  | 2946|
| \(M_4\) | 64     | 4  | 8  | 456   | 3025| 6   | 1058 | 13824|
| \(M_5\) | 160    | 5  | 6  | 809   | 7665| 17  | 2723 | 34057|
| \(M_6\) | 384    | 6  | 11 | 1383  | 12870| 25  | 6250 | 90370|
| \(M_7\) | 896    | 7  | 11 | 2087  | 24156| 52  | 14627 | 226114|
| \(M_8\) | 2048   | 8  | 13 | 3289  | 41732| 100 | 34042 | 651678|
| bbara   | 7      | 2  | 3  | 167   | 1049| 3   | 216  | 1535|
| mark1   | 202    | 8^*| 22 | 13027 | 117735| 67  | 15192 | 252874|
| keyb    | 41     | 2  | 25 | 12464 | 153899| 24  | 6024 | 265805|
| ex3     | 28     | 2  | 24 | 1133  | 9042 | 18  | 878  | 91494|
REFERENCES

[1] F. Vaandrager, “Model learning,” Communications of the ACM, vol. 60, no. 2, pp. 86–95, 2017.

[2] P. Fiterau-Broştean, R. Janssen, and F. Vaandrager, “Combining model learning and model checking to analyze TCP implementations,” in International Conference on Computer Aided Verification. Springer, 2016, pp. 454–471.

[3] M. Schuts, J. Hooman, and F. Vaandrager, “Refactoring of legacy software using model learning and equivalence checking: an industrial experience report,” in International Conference on Integrated Formal Methods. Springer, 2016, pp. 311–325.

[4] R. Alur, P. Černý, P. Madhusudan, and W. Nam, “Synthesis of interface specifications for java classes,” ACM SIGPLAN Notices, vol. 40, no. 1, pp. 98–109, 2005.

[5] M. Isberner, F. Howar, and B. Steffen, “The open-source LearnLib - A framework for active automata learning,” in International Conference on Computer Aided Verification. Springer, 2015, pp. 487–495.

[6] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, “libalf: The automata learning framework,” in International Conference on Computer Aided Verification. Springer, 2010, pp. 360–364.

[7] R. L. Rivest and R. E. Schapire, “Diversity-based inference of finite automata,” Journal of the ACM (JACM), vol. 41, no. 3, pp. 555–589, 1994.

[8] D. Angluin, “Learning regular sets from queries and counterexamples,” Information and computation, vol. 75, no. 2, pp. 87–106, 1987.

[9] G. van Heerdt, M. Sammartino, and A. Silva, “CALF: Categorical automata learning framework,” 2017, under submission, arXiv:1704.05676.

[10] G. van Heerdt, “An abstract automata learning framework,” Master’s thesis, Radboud University, 2016.

[11] M. Isberner, F. Howar, and B. Steffen, “The TTT algorithm: A redundancy-free approach to active automata learning,” in RV, 2014, pp. 307–322.

[12] R. M. Hierons and U. C. Türker, “Incomplete distinguishing sequences for finite state machines,” The Computer Journal, vol. 58, no. 11, pp. 3089–3113, 2015.

[13] J. Rot, “Coalgebraic minimization of automata by initiality and finality,” Electronic Notes in Theoretical Computer Science, vol. 325, pp. 253–276, 2016.

[14] F. Bonchi, M. M. Bonsangue, H. H. Hansen, P. Panangaden, J. J. Rutten, and A. Silva, “Algebra-coalgebra duality in Brzozowski’s minimization algorithm,” ACM Transactions on Computational Logic (TOCL), vol. 15, no. 1, p. 3, 2014.

[15] C. L. Nehaniv, “Algebraic models for understanding: coordinate systems and cognitive empowerment,” in Second International Conference on Cognitive Technology, 1997. Humanizing the Information Age. Proceedings. IEEE, 1997, pp. 147–162.