Automata Cascades: Expressivity and Sample Complexity

Alessandro Ronca[1], **Nadezda Alexandrovna Knorozova**[2,3], **Giuseppe De Giacomo**[1,4]
[1]DIAG, Sapienza University of Rome
[2]RelationalAI
[3]IFI, University of Zurich
[4]Computer Science Department, University of Oxford
ronca@diag.uniroma1.it, nadezda.knorozova@relational.ai, giuseppe.degiacomo@cs.ox.ac.uk

Abstract

Every automaton can be decomposed into a cascade of basic *prime* automata. This is the Prime Decomposition Theorem by Krohn and Rhodes. Guided by this theory, we propose *automata cascades* as a structured, modular, way to describe automata as complex systems made of many components, each implementing a specific functionality. Any automaton can serve as a component; using specific components allows for a fine-grained control of the expressivity of the resulting class of automata; using prime automata as components implies specific expressivity guarantees. Moreover, specifying automata as cascades allows for describing the sample complexity of automata in terms of their components. We show that the sample complexity is linear in the number of components and the maximum complexity of a single component, modulo logarithmic factors. This opens to the possibility of learning automata representing large dynamical systems consisting of many parts interacting with each other. It is in sharp contrast with the established understanding of the sample complexity of automata, described in terms of the overall number of states and input letters, which implies that it is only possible to learn automata where the number of states is linear in the amount of data available. Instead our results show that one can learn automata with a number of states that is exponential in the amount of data available.

Introduction

Automata are fundamental in computer science. They are one of the simplest models of computation, with the expressive power of regular languages, placed at the bottom of the Chomsky hierarchy. They are also a mathematical model of finite-state dynamical systems. In learning applications, automata allow for capturing targets that exhibit a time-dependent behaviour; namely, functions over sequences such as time-series, traces of a system, histories of interactions between an agent and its environment. Automata are typically viewed as *state diagrams*, where states and transitions are the building blocks of an automaton, cf. (Hopcroft and Ullman 1979). Accordingly, classes of automata are described in terms of the number of states and input letters. Learning from such classes requires an amount of data that is linear in the number of states and letters (Ishigami and Tani 1997). Practically it means that, in order to learn large dynamical systems made of many components, the amount of data required is exponential in the number of components, as the number of states of a system will typically be exponential in the number of its stateful components.

We propose *automata cascades* as a structured, modular, way to describe automata as complex systems made of many *components* connected in an acyclic way. Our cascades are strongly based on the *theory of Krohn and Rhodes*, which says that every automaton can be decomposed into a cascade of basic components called *prime automata* (Krohn and Rhodes 1965). Conversely, the theory can be seen as prescribing which components to use in order to build certain classes of automata, and hence obtain a certain expressivity. For example, we can cascade so-called *flip-flop automata* in order to build all noncounting automata, and hence obtain the expressivity of well-known logics such as monadic first-order logic on finite linearly-ordered domains (McNaughton and Papert 1971) and the linear temporal logic on finite traces LTL_f (De Giacomo and Vardi 2013).

We focus on cascades as a means to learn automata. Our cascades are designed for a fine control of their sample complexity. We show that—ignoring logarithmic factors—*the sample complexity of automata cascades is at most linear in the product of the number of components and the maximum complexity of a single component*. Notably, the complexity of a single component does not grow with the number of components in the cascade. We carry out the analysis both in the setting where classes of automata cascades are finite, and in the more general setting where they can be infinite. For both cases, we obtain bounds of the same shape, with one notable difference that for infinite classes we incur a logarithmic dependency on the maximum length of a string. Overall, our results show that the sample complexity of automata can be decoupled from the the number of states and input letters. Rather, it can be described in terms of the components of a cascade capturing the automaton. Notably, the number of states of such an automaton can be exponential in the number of components of the cascade.

We see the opportunity for cascades to unlock a greater potential of automata in learning applications. On one hand, automata come with many favourable, well-understood, theoretical properties, and they admit elegant algorithmic solutions. On the other hand, the existing automata learning

algorithms have a complexity that depends directly on the number of states. This hurts applications where the number of states grows very fast, such as non-Markov reinforcement learning (Toro Icarte et al. 2018; De Giacomo et al. 2019; Gaon and Brafman 2020; Brafman and De Giacomo 2019; Abadi and Brafman 2020; Xu et al. 2020; Neider et al. 2021; Jothimurugan et al. 2021; Ronca and De Giacomo 2021; Ronca, Paludo Licks, and De Giacomo 2022). Given our favourable sample complexity results, automata cascades have a great potential to extend the applicability of automata learning in large complex settings.

Before concluding the section, we introduce our running example, that is representative of a class of tasks commonly considered in reinforcement learning. It is based on an example from (Andreas, Klein, and Levine 2017).

Example 1. *Consider a Minecraft-like domain, where an agent has to build a bridge by first collecting some raw materials, and then using a factory. The agent has two options. The first option is to complete the tasks* $\{\text{getWood}, \text{getIron}, \text{getFire}\}$ *in any order, and then* $\{\text{useFactory}\}$*. The second option is to complete* $\{\text{getSteel}\}$*, and then* $\{\text{useFactory}\}$*.*

A formal description of the example as well as full proofs of all our technical results are given in the extended version (Ronca, Knorozova, and De Giacomo 2022).

Preliminaries

Functions. For $f : X \to Y$ and $g : Y \to Z$, their *composition* $f \circ g : X \to Z$ is defined as $(f \circ g)(x) = g(f(x))$. For $f : X \to Y$ and $h : X \to Z$, their *cross product* $f \times h : X \to Y \times Z$ is defined as $(f \times h)(x) = \langle f(x), h(x) \rangle$. A class of functions is *uniform* if all functions in the class have the same domain and codomain. Given a tuple $t = \langle x_1, \ldots, x_n \rangle$, and a subset $J \subseteq [1, n]$, the *projection* $\pi_J(t)$ is $\langle x_{j_1}, \ldots, x_{j_m} \rangle$ where $j_1, \ldots, j_m$ is the sorted sequence of elements of J. Furthermore, π^a_m denotes the class of all projections π_J for J a subset of $[1, a]$ of cardinality m. We write I for the *identity function*, and $\log$ for $\log_2$.

String Functions and Languages. An *alphabet* Σ is a set of elements called *letters*. A *string* over Σ is an expression $\sigma_1 \ldots \sigma_\ell$ where each letter σ_i is from Σ. The *empty string* is denoted by ε. The set of all strings over Σ is denoted by Σ^*. A *factored alphabet* is of the form X^a for some set X called the *domain* and some integer $a \geq 1$ called the *arity*. A *string function* is of the form $f : \Sigma^* \to \Gamma$ for Σ and Γ alphabets. *Languages* are a special case of string functions; namely, when $f : \Sigma \to \{0, 1\}$ is an indicator function, it can be equivalently described by the set $\{x \in \Sigma^* \mid f(x) = 1\}$, that is called a *language*.

Learning Theory

We introduce the problem of learning, following the classical perspective of statistical learning theory (Vapnik 1998).

Learning Problem. Consider an input domain X and an output domain Y. For example, in the setting where we classify strings over an alphabet Σ, the input domain X is Σ^* and the output domain Y is $\{0, 1\}$. The *problem of learning* is that of choosing, from an *admissible class* $\mathcal{F}$ of functions from X to Y, a function f that best approximates an unknown *target function* $f_0 : X \to Y$, not necessarily included in $\mathcal{F}$. The quality of the approximation of f is given by the overall discrepancy of f with the target f_0. On a single domain element x, the discrepancy between $f(x)$ and $f_0(x)$ is measured as $L(f(x), f_0(x))$, for a given *loss function* $L : Y \times Y \to \{0, 1\}$. The overall discrepancy is the expectation $\mathbb{E}[L(f(x), f_0(x))]$ with respect to an underlying probability distribution P, and it is called the *risk* of the function, written $R(f)$. Then, the goal is to choose a function $f \in \mathcal{F}$ that minimises the risk $R(f)$, when the underlying probability distribution P is unknown, but we are given a *sample* Z_ℓ of ℓ i.i.d. elements $x_i \in X$ drawn according to $P(x_i)$ together with their labels $f_0(x_i)$; specifically, $Z_\ell = z_1, \ldots, z_\ell$ with $z_i = \langle x_i, f_0(x_i) \rangle$.

Sample Complexity. We would like to establish the minimum sample size ℓ sufficient to identify a function $f \in \mathcal{F}$ such that

$$R(f) - \min_{f \in \mathcal{F}} R(f) \leq \epsilon$$

with probability at least $1 - \eta$. We call such ℓ the sample complexity of $\mathcal{F}$, and we write it as $S(\mathcal{F}, \epsilon, \eta)$. When ϵ and η are considered fixed, we write it as $S(\mathcal{F})$.

Sample Complexity Bounds for Finite Classes. When the set of admissible functions $\mathcal{F}$ is finite, its sample complexity can be bounded in terms of its cardinality, cf. (Shalev-Shwartz and Ben-David 2014). In particular,

$$S(\mathcal{F}, \epsilon, \eta) \in O\big((\log |\mathcal{F}| - \log \eta)/\epsilon^2\big).$$

Then, for fixed ϵ and η, the sample complexity $S(\mathcal{F})$ is $O(\log |\mathcal{F}|)$, and hence finite classes can be compared in terms of their cardinality.

Automata

This section introduces basic notions of automata theory, with some inspiration from (Ginzburg 1968; Maler 1990).

An automaton is a mathematical description of a stateful machine that returns an output letter on every input string. At its core lies the mechanism that updates the internal state upon reading an input letter. This mechanism is captured by the notion of semiautomaton. An n-state *semiautomaton* is a tuple $D = \langle \Sigma, Q, \delta, q_{\text{init}} \rangle$ where: Σ is an alphabet called the *input alphabet*; Q is a set of n elements called *states*; $\delta : Q \times \Sigma \to Q$ is a function called *transition function*; $q_{\text{init}} \in Q$ is called *initial state*. The transition function is recursively extended to non-empty strings as $\delta(q, \sigma_1 \sigma_2 \ldots \sigma_m) = \delta(\delta(q, \sigma_1), \sigma_2 \ldots \sigma_m)$, and to the empty string as $\delta(q, \varepsilon) = q$. The result of executing semiautomaton D on an input string is $D(\sigma_1 \ldots \sigma_m) = \delta(q_{\text{init}}, \sigma_1 \ldots \sigma_m)$. We also call such function $D : \Sigma^* \to Q$ the function implemented by the semiautomaton.

Automata are obtained from semiautomata by adding an output function. An n-state *automaton* is a tuple $A = \langle \Sigma, Q, \delta, q_{\text{init}}, \Gamma, \theta \rangle$ where: $D_A = \langle \Sigma, Q, \delta, q_{\text{init}} \rangle$ is a semiautomaton, called the *core semiautomaton* of A; Γ is an alphabet called *output alphabet*, $\theta : Q \times \Sigma \to \Gamma$ is called

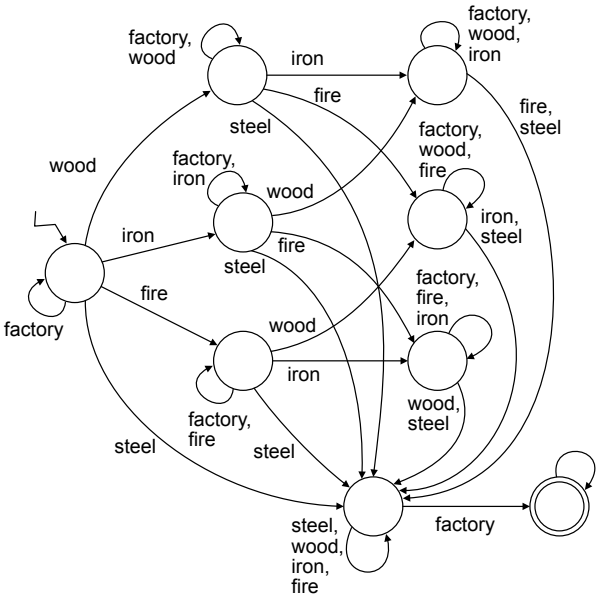

Figure 1: State diagram of the automaton for Example 2.

output function. An *acceptor* is a special kind of automaton where the output function is an indicator function $\theta : Q \times \Sigma \to \{0, 1\}$. The result of executing autmaton A on an input string is $A(\sigma_1 \dots \sigma_m) = \theta(D_A(\sigma_1 \dots \sigma_{m-1}), \sigma_m)$. We also call such function $A : \Sigma^* \to \Gamma$ the function implemented by the automaton. The language *recognised* by an acceptor is the set of strings on which it returns 1. When two automata A_1 and A_2 implement the same function, we say that A_1 *is captured by* A_2, or equivalently that A_2 is captured by A_1.

The *expressivity* of a class of automata is the set of functions they implement. Thus, for acceptors, it is the set of languages they recognise. The expressivity of all acceptors is the *regular languages* (Kleene 1956), i.e., the languages that can be specified by regular expressions. We will often point out the expressivity of acceptors because the classes of languages they capture are widely known.

Example 2. *The automaton for our running example reads traces generated by the agent while interacting with the environment. The input alphabet is* $\Sigma = \{\text{blank}, \text{wood}, \text{iron}, \text{fire}, \text{steel}, \text{factory}\}$*, where* blank *describes that no relvant event happened. The automaton returns* 1 *on traces where the agent has completed the task. The state diagram is depicted in Figure 1, where transitions for* blank *are omitted and they always yield the current state. The output is* 1 *on all transitions entering the double-lined state, and* 0 *otherwise.*

The automaton in the example has to keep track of the subset of tasks completed so far, requiring one state per subset. In general, the number of states can grow exponentially with the number of tasks.

Existing Sample Complexity Results

Classes of automata are typically defined in terms of the cardinality k of the input alphabet (assumed to be finite) and the number n of states. The existing result on the sample complexity of automata is for such a family of classes.

Theorem 1 (Ishigami and Tani, 1997)**.** *Let $\mathcal{A}(k, n)$ be the class of n-state acceptors over the input alphabet $[1, k]$. Then, the sample complexity of $\mathcal{A}(k, n)$ is $\Theta(k \cdot n \cdot \log n)$.*[1]

Consequently, learning an acceptor from the class of all acceptors with k input letters and n states requires an amount of data that is at least $k \cdot n$. Such a dependency is also observed in the existing automata learning algorithms, e.g., (Angluin 1987; Ron, Singer, and Tishby 1996, 1998; Clark and Thollard 2004; Palmer and Goldberg 2007; Balle, Castro, and Gavaldà 2013, 2014). More recently, there has been an effort in overcoming the direct dependency on the cardinality k of the input alphabet, through *symbolic automata* (Mens and Maler 2015; Maler and Mens 2017; Argyros and D'Antoni 2018), but their sample complexity has not been studied.

Automata Cascades

We present the formalism of automata cascades, strongly based on the cascades from (Krohn and Rhodes 1965)—see also (Ginzburg 1968; Maler 1990; Dömösi and Nehaniv 2005). The novelty of our formalism is that every cascade component is equipped with mechanisms for processing inputs and outputs. This allows for (*i*) controlling the complexity of components as the size of the cascade increases; and (*ii*) handling large (and even infinite) input alphabets.

Definition 1. *An* automata cascade *is a sequence of automata $A_1 \ltimes \dots \ltimes A_d$ where each A_i is called a* component *of the cascade and it is of the form*

$$\langle \Sigma_1 \times \dots \times \Sigma_i, Q_i, \delta_i, q_i^{\text{init}}, \Sigma_{i+1}, \theta_i \rangle.$$

The function implemented by the cascade is the one implemented by the automaton $\langle \Sigma_1, Q, \delta, q^{\text{init}}, \Sigma_{d+1}, \theta \rangle$ having set of states $Q = Q_1 \times \dots \times Q_d$, initial state $q_{\text{init}} = \langle q_1^{\text{init}}, \dots, q_d^{\text{init}} \rangle$, transition and output functions defined as

$$\delta(\langle q_1, \dots, q_d \rangle, \sigma) = \langle \delta_1(q_1, \sigma_1), \dots, \delta_d(q_d, \sigma_d) \rangle,$$
$$\theta(\langle q_1, \dots, q_d \rangle, \sigma) = \theta_d(q_d, \sigma_d),$$

where each component reads the recursively-defined input

$$\sigma_1 = \sigma, \text{ and } \sigma_{i+1} = \langle \sigma_i, \theta_i(q_i, \sigma_i) \rangle.$$

A cascade is simple *if $\theta_i(q, \sigma) = q$, for every $i \in [1, d-1]$. An* acceptor cascade *is a cascade where $\Gamma_{d+1} = \{0, 1\}$.*

The components of a cascade are arranged in a sequence. Every component reads the input and output of the preceding component, and hence, recursively, it reads the external input together with the output of all the preceding components. The external input is the input to the first component, and the overall output is the one of the last component.

As components of a cascade, we consider automata on factored alphabets that first apply a *projection* operation on their input, then apply a map to a smaller *internal alphabet*, and finally transition based on the result of the previous operations.

[1] The original result is equivalently stated in terms of the VC dimension, introduced later in this paper.

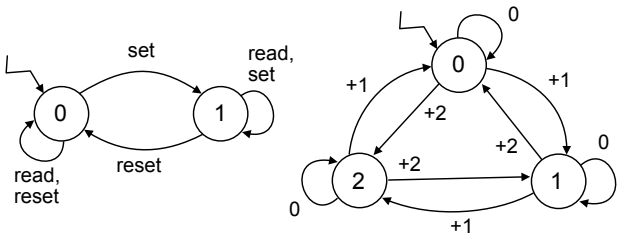

Figure 2: State diagrams of some of the simplest prime automata: a flip-flop on the left, and a 3-counter on the right.

Definition 2. *An n-state* automaton *is a tuple $A = \langle X^a, J, \Pi, \phi, Q, \delta, q_{\text{init}}, \Gamma, \theta\rangle$ where X^a is the factored input alphabet; $J \subseteq [1, a]$ is the* dependency set *and its cardinality m is the* degree of dependency*; Π is the finite* internal alphabet*; $\phi : X^m \to \Pi$ is an* input function *that operates on the projected input tuples; Q is a set of n states; $\delta : Q \times \Pi \to Q$ is the transition function on the internal letters; $q_{\text{init}} \in Q$ is the initial state; Γ is the output alphabet; and $\theta : Q \times X^m \to \Gamma$ is an output function that operates on the projected input tuples. The automaton induced by A is the automaton $A' = \langle \Sigma, Q, \delta_{J,\phi}, q_{\text{init}}, \Gamma, \theta_J\rangle$ where the input alphabet is $\Sigma = X^a$, the transition function is $\delta_{J,\phi}(q, \sigma) = \delta(q, \phi(\pi_J(\sigma)))$, and the output function is $\theta_J(q, \sigma) = \theta(q, \pi_J(\sigma))$. The core semiautomaton of A is the core semiautomaton of A'. The string function implemented by A is the one implemented by the induced automaton.*

The above definition adds two key aspects to the standard definition of automaton. First, the *projection operation* π_J, that allows for capturing the dependencies between the components in a cascade. Although every component receives the output of all the preceding components, it may use only some of them, and hence the others can be projected away. The dependency set J corresponds to the indices of the input tuple that are relevant to the component. Second, the *input function* ϕ, that maps the result of the projection operation to an internal letter. The rationale is that many inputs trigger the same transitions, and hence they can be mapped to the same internal letter. This particularly allows for decoupling the size of the core semiautomaton from the cardinality of the input alphabet—in line with the mechanisms of symbolic automata (Maler and Mens 2017; Argyros and D'Antoni 2018).

Expressivity of Automata Cascades

Cascades can be built out of any set of components. However, the theory by Krohn and Rhodes identifies a set of *prime automata* that is a sufficient set of components to build cascades, as it allows for capturing all automata. They are, in a sense, the building blocks of automata. Moreover, using only some prime automata, we obtain specialised expressivity results.

Prime Components

Prime automata are partitioned into two classes. The *first class of prime automata* are flip-flops, a kind of automaton that allows for storing one bit of information.

Definition 3. *A* flip-flop *is a two-state automaton $\langle X^a, J, \Pi, \phi, Q, \delta, q_{\text{init}}, \Gamma, \theta\rangle$ where $\Pi = \{\text{set}, \text{reset}, \text{read}\}$, $Q = \{0, 1\}$ and the transition function satisfies the following three identities:*

$$\delta(q, \text{read}) = q, \quad \delta(q, \text{set}) = 1, \quad \delta(q, \text{reset}) = 0.$$

We capture the task of our running example with a cascade where each task is captured exactly by a flip-flop.

Example 3. *The sequence task of our running example is captured by the cascade*

$$A_{\text{wood}} \ltimes A_{\text{iron}} \ltimes A_{\text{fire}} \ltimes A_{\text{steel}} \ltimes A_{\text{factory}}$$

where each component is a flip-flop that outputs its current state. The diagram for the cascade is shown in Figure 3, where getWood *corresponds to A_{wood}, and similarly for the other components. All components read the input, and only A_{factory} also reads the output of the other components. Thus, the dependency set of A_{wood}, A_{iron}, A_{fire}, and A_{steel} is the singleton $\{1\}$, and the dependency set of A_{factory} is $\{1, 2, 3, 4, 5\}$—note that the indices correspond to positions of the components in the cascade. Then, A_{wood} has input function $\phi_{\text{wood}}(x)$ that returns* set *if $x =$ wood, and returns* read *otherwise. Similarly, A_{iron}, A_{fire}, and A_{steel}. Instead, the component A_{factory} has input function $\phi_{\text{factory}}(x, \mathit{wood}, \mathit{iron}, \mathit{fire}, \mathit{steel})$ that returns* set *if*

$$(x = \text{factory}) \wedge [(\mathit{wood} \wedge \mathit{iron} \wedge \mathit{fire}) \vee \mathit{steel}]$$

and returns read *otherwise.*

The *second class of prime automata* is a class of automata that have a correspondence with simple groups from group theory. Their general definition is beyond the scope of this paper. For that, see (Ginzburg 1968). Here we present the class of prime counters, as a subclass that seems particularly relevant from a practical point of view.

Definition 4. *An* n-counter *is an n-state automaton $\langle X^a, J, \Pi, \phi, Q, \delta, q_{\text{init}}, \Gamma, \theta\rangle$ where $\Pi = Q = [0, n-1]$, and the transition function satisfies the following identity:*

$$\delta(i, j) = i + j \pmod n.$$

An n-counter is prime *if n is a prime number.*

An n-counter implements a counter modulo n. The internal letters correspond to numbers that allow for reaching any value of the counter in one step. In particular, they also allow for implementing the functionality of decreasing the counter, e.g., adding $n - 1$ amounts to subtracting 1. Note also that the internal letter 0 plays the same role as *read* does in a flip-flop. When we are interested just in counting—i.e., increment by one—the modulo stands for the overflow due to finite memory. On the other hand, we might actually be interested in counting modulo n; for instance, to capture periodic events such as 'something happens every 24 hours'. A 3-counter, that is a prime counter, is depicted in Figure 2.

Example 4. *Resuming our running example, say that now, in order to use the factory, we need (i)* 13 *pieces of wood,* 5 *pieces of iron, and fire, or alternatively (ii)* 7 *pieces of*

steel. From the cascade in Example 3, it suffices to change the cascade components $A_{\text{wood}}, A_{\text{iron}}, A_{\text{steel}}$ into counters (e.g., 16-counters) and change the input function of A_{factory} so that $\phi_{\text{factory}}(x, wood, iron, fire, steel)$ returns set *if*

$$(x = \text{factory}) \wedge$$
$$\big[\big((wood \geq 13) \wedge (iron \geq 5) \wedge fire\big) \vee (steel \geq 7)\big],$$

and it returns read *otherwise. The rest is left unchanged. The cascade, despite its simplicity, corresponds to an automaton that has over* 700 *states. Furthermore, suppose that we need to learn to detect* wood, iron, fire, steel *from video frames represented as vectors of $\mathbb{R}^a$. It suffices to replace the input function of the corresponding components with a function over $\mathbb{R}^a$ such as a neural network.*

Expressivity Results

A key aspect of automata cascades is their expressivity. As an immediate consequence of the Krohn-Rhodes theorem (Krohn and Rhodes 1965)—see also (Ginzburg 1968; Maler 1990; Dömösi and Nehaniv 2005)—we have that simple cascades of prime automata capture all automata, and simple cascades of flip-flops capture the so-called *group-free automata*.

Theorem 2. *Every automaton is captured by a simple cascade of prime automata. Furthermore, every group-free automaton is captured by a simple cascade of flip-flops. The converse of both claims holds as well.*

Group-free automata are important because they capture *noncounting automata*, c.f. (Ginzburg 1968), whose expressivity is the *star-free regular languages* (Schützenberger 1965)—i.e., the languages that can be specified by regular expressions without using the Kleene star but using complementation. This is the expressivity of well-known logics such as *monadic first-order logic on finite linearly-ordered domains* (McNaughton and Papert 1971), and the *linear temporal logic on finite traces* LTL_f (De Giacomo and Vardi 2013). This observation, together with the fact that the expressivity of all acceptors is the regular languages, allows us to derive the following theorem from the one above.

Theorem 3. *The expressivity of simple acceptor cascades of prime automata is the regular languages. The expressivity of simple acceptor cascades of flip-flops is the star-free regular languages.*

Sample Complexity of Automata Cascades

We study the sample complexity of classes of automata (and cascades thereof) built from given classes of input functions, semiautomata, and output functions. This allows for a fine-grained specification of automata and cascades.

Definition 5. *The class $\mathcal{A}(\Phi, \Delta, \Theta)$ over a given input alphabet X^a consists of each automaton with input function from a given class Φ, core semiautomaton from a given class Δ, and output function from a give class Θ. The classes Φ, Δ, Θ are uniform, and hence all automata in $\mathcal{A}(\Phi, \Delta, \Theta)$ have the same degree of dependency m, same internal alphabet Π, and same output alphabet Γ; hence, sometimes we write $\mathcal{A}(\Phi, \Delta, \Theta; a, m)$ and $\mathcal{A}(\Phi, \Delta, \Theta; a, m, \Pi, \Gamma)$ to have such quantities at hand.*

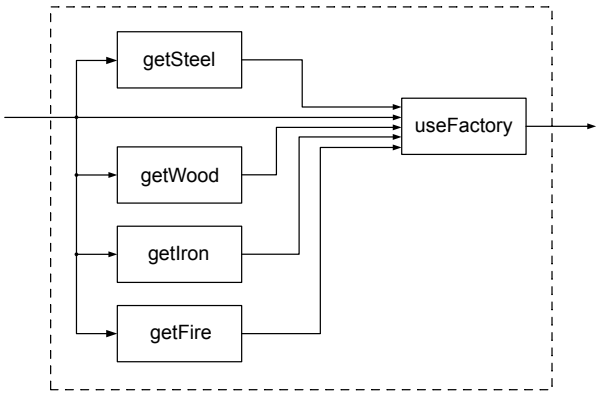

Figure 3: Diagram for the cascade of the running example.

Results for Finite Classes of Cascades

The following two theorems establish upper bounds on the cardinality and sample complexity of finite classes of automata, and finite classes of cascades, respectively. The results are obtained by counting the number of ways in which we can instantiate the parts of a single automaton.

Theorem 4. *The cardinality of a class of automata $\mathcal{A} = \mathcal{A}(\Phi, \Delta, \Theta; a, m)$ is bounded as*

$$|\mathcal{A}| \leq |\pi_m^a| \cdot |\Phi| \cdot |\Delta| \cdot |\Theta|\,,$$

and its sample complexity is asymptotically bounded as

$$S(\mathcal{A}) \in O\big(\log|\pi_m^a| + \log|\Phi| + \log|\Delta| + \log|\Theta|\,\big).$$

Theorem 5. *The cardinality of a class of automata cascades $\mathcal{C} = \mathcal{A}_1 \ltimes \cdots \ltimes \mathcal{A}_d$ where the automata classes are $\mathcal{A}_i = \mathcal{A}(\Phi_i, \Delta_i, \Theta_i; a_i, m_i)$ is bounded as*

$$|\mathcal{C}| \leq \prod_{i=1}^{d} |\pi_{m_i}^{a_i}| \cdot |\Phi_i| \cdot |\Delta_i| \cdot |\Theta_i|\,,$$

and its sample complexity is asymptotically bounded as

$$S(\mathcal{C}) \in O\Big(d \cdot (\log|\pi_m^a| + \log|\Phi| + \log|\Delta| + \log|\Theta|)\Big),$$

where dropping the indices denotes the maximum.

Consequently, the complexity of a cascade is bounded by the product of the number d of components and a second factor that bounds the complexity of a single component.

Aspects and Implications of The Results

The term $\log|\pi_m^a|$, accounting for the projection functions, ranges from 0 to $\min(m, a_1 + d - m) \cdot \log(a_1 + d)$ where a_1 is the input arity of the first component, and hence of the external input. In particular, it is minimum when we allow each component to depend on all or none of its preceding components, and it is maximum when each component has to choose half of its preceding components as its dependencies.

The term $\log|\Phi|$ plays an important role, since the set of input functions has to be sufficiently rich so as to map the external input and the outputs of the preceding components to

the internal input. First, its cardinality can be controlled by the degree of dependency m; for instance, taking all Boolean functions yields $\log|\Phi| = 2^m$. Notably, it does not depend on d, and hence, the overall sample complexity grows linearly with the number of cascade components as long as the degree of dependency m is bounded. Second, the class of input functions can be tailored towards the application at hand. For instance, in our running example, input functions are chosen from a class Φ for which $\log|\Phi|$ is linear in the number of tasks—see Example 5 below.

The term $\log|\Delta|$ is the contribution of the *number of semiautomata*—note that the number of letters and states of each semiautomaton plays no role. Interestingly, very small classes of semiautomata suffice to build very expressive cascades, by the results in Section . In general, it is sufficient to build Δ out of prime components. But we can also include other semiautomata implementing some interesting functionality.

The term $\log|\Theta|$ admits similar considerations as $\log|\Phi|$. It is worth noting that one can focus on simple cascades, by the results in Section , where output functions of all but last component are fixed. Then, the contribution to the sample complexity is given by the class of output functions of the last component.

Overall, the sample complexity depends linearly on the number d of components, if we ignore logarithmic factors. Specifically, the term $\log|\pi_m^a|$ has only a logarithmic dependency on d, and the other terms $\log|\Phi|$, $\log|\Delta|$, and $\log|\Theta|$ are independent of d.

Corollary 1. *Let us recall the quantities from Theorem 5, and fix the quantities a, m, Φ, Δ, and Θ. Then, the sample complexity of $\mathcal{C}$ is bounded as $S(\mathcal{C}) \in O(d \cdot \log d)$.*

In the next example we instantiate a class of cascades for our running example, and derive its sample complexity.

Example 5. *The cascade described in Example 3 has one component per task, and all components have the same output function and semiautomaton. The input function ϕ_{factory} is 2-term monotone DNF over 9 propositional variables. Every other component has an input function that is 1-term monotone DNF over 5 propositional variables. Using these observations, we can design a class of cascades for similar sequence tasks where the goal task depends on two groups of arbitrary size, having d basic tasks overall. Such a class will consist of cascades of d components. For every $i \in [1, d-1]$, the class of input functions Φ_i is the class of 1-term monotone DNF over d variables; and Φ_d is the class of 2-term monotone DNF over $2d-1$ variables. For all $i \in [1, d]$, Δ_i is a singleton consisting of a flip-flop semiautomaton; and Θ_i is a singleton consisting of the function that returns the state. The cardinality of Φ_i is $e^2 \cdot 2^{(4d-4)}$, and hence its logarithm is less than $4d$—see Corollary 5 of (Schmitt 2004). By Theorem 5, considering that we have d cascade components, we obtain that our class of cascades has sample complexity $O(d^2)$. At the same time, the minimum automaton for the considered family of sequence tasks has $\Omega(2^d)$ states. Going back to the bound of Theorem 1, if we had to learn from the class of all automata with 2^d states, we would incur a sample complexity exponential in the number of tasks.*

Learning Theory for Infinite Classes

Infinite classes of cascades naturally arise when the input alphabet is infinite. In this case, one may consider to pick input and output functions from an infinite class. For instance, the class of all threshold functions over the domain of integers, or as in Example 4 the neural networks over the vectors of real numbers.

When considering infinite classes of functions, the sample complexity bounds based on cardinality become trivial. However, even when the class is infinite, the number of functions from the class that can be distinguished on a given sample is finite, and the way it grows as a function of the sample size allows for establishing sample complexity bounds. In turn, such *growth* can be bounded in terms of the *dimension* of the class of functions, a single-number characterisation of its complexity. Next we present these notions formally.

Growth and Dimension. Let $\mathcal{F}$ be a class of functions from a set X to a finite set Y. Let $X_\ell = x_1, \ldots, x_\ell$ be a sequence of ℓ elements from X. The set of *patterns* of a class $\mathcal{F}$ on X_ℓ is

$$\mathcal{F}(X_\ell) = \{\langle f(x_1), \ldots, f(x_\ell)\rangle \mid f \in \mathcal{F}\},$$

and the *number of distinct patterns* of class $\mathcal{F}$ on X_ℓ is $N(\mathcal{F}, X_\ell) = |\mathcal{F}(X_\ell)|$. The *growth* of $\mathcal{F}$ is

$$G(\mathcal{F}, \ell) = \sup_{X_\ell} N(\mathcal{F}, X_\ell).$$

The growth of $\mathcal{F}$ can be bounded using its *dimension*, written as $\dim(\mathcal{F})$. When $|Y| = 2$, we define the dimension of $\mathcal{F}$ to be its *VC dimension* (Vapnik and Chervonenkis 1971)—see also (Vapnik 1998). It is the largest integer h such that $G(\mathcal{F}, h) = 2^h$ and $G(\mathcal{F}, h+1) < 2^{h+1}$ if such an h exists, and infinity otherwise. When $|Y| > 2$, we define the dimension of $\mathcal{F}$ to be its *graph dimension* (Natarajan 1989; Haussler and Long 1995). It is defined by first binarising the class of functions. For a given function $f : X \to Y$, its binarisation $f_{\text{bin}} : X \times Y \to \{0, 1\}$ is defined as $f_{\text{bin}}(x, y) = \mathbf{1}[f(x) = y]$. The *binarisation* of $\mathcal{F}$ is $\mathcal{F}_{\text{bin}} = \{f_{\text{bin}} \mid f \in \mathcal{F}\}$. Then, the graph dimension of $\mathcal{F}$ is defined as the VC dimension of its binarisation $\mathcal{F}_{\text{bin}}$.

The growth of $\mathcal{F}$ can be bounded in terms of its dimension (Haussler and Long 1995), as follows:

$$G(\mathcal{F}, \ell) \leq (e \cdot \ell \cdot |Y|)^{\dim(\mathcal{F})}.$$

Sample Complexity. The sample complexity can be bounded in terms of the dimension, cf. (Shalev-Shwartz and Ben-David 2014). In particular,

$$S(\mathcal{F}, \epsilon, \eta) \in O\left((\dim(\mathcal{F}) \cdot \log|Y| - \log\eta)/\epsilon^2\right).$$

For fixed ϵ, η, and Y, the sample complexity is $O(\dim(\mathcal{F}))$, and hence arbitrary classes over the same outputs can be compared in terms of their dimension.

Results for Infinite Classes of Cascades

We generalise our sample complexity bounds to infinite classes of automata and cascades. The bound have the same shape of the bounds derived in Section for finite classes,

with the dimension replacing the logarithm of the cardinality. One notable difference is the logarithmic dependency on the maximum length M of a string. It occurs due to the (stateless) input and output functions. Their growth is on single letters, regardless of the way they are grouped into strings. Thus, the growth on a sample of ℓ strings is the growth on a sample of $\ell \cdot M$ letters.

The bounds are derived using a functional description of automata and cascades. It is a shift of perspective, from stateful machines that process one letter at a time, to black-boxes that process an entire input string at once. We first introduce function constructors to help us to build such descriptions.

Definition 6 (Function constructors)**.** *For $f : \Sigma \to \Gamma$, $f^* : \Sigma^* \to \Gamma$ is defined as $f^*(\sigma_1 \dots \sigma_n) = f(\sigma_n)$. For $g : \Sigma^* \to \Gamma$, $\overline{g} : \Sigma^* \to \Gamma^*$ is defined as $\overline{g}(\sigma_1 \dots \sigma_n) = g(\sigma_1) \dots g(\sigma_1 \dots \sigma_n)$; furthermore, $g^\triangleleft : \Sigma^* \to \Gamma^*$ is defined as $g^\triangleleft(\sigma_1 \dots \sigma_n) = g(\sigma_1 \dots \sigma_{n-1})$.*

Lemma 1. *The function A implemented by an automaton $\langle X^a, J, \Pi, \phi, Q, \delta, q_{\text{init}}, \Gamma, \theta \rangle$ can be expressed as*

$$A = \overline{\pi_J^*} \circ ((\overline{\phi^*} \circ D^\triangleleft) \times I^*) \circ \theta,$$

where D is the function implemented by the semiautomaton $\langle \Pi, Q, \delta, q_{\text{init}} \rangle$.

Note that I^* propagates the projected input to the output function. Also note that the output function reads the state before the last update. The functional description allows us to bound the growth, by making use of the fact that the growth of composition and cross product of two classes of functions is upper bounded by the product of their respective growths. From there, we derive the dimension, and the sample complexity.

Theorem 6. *Let $\mathcal{A}$ be a class $\mathcal{A}(\Phi, \Delta, \Theta; a, m, \Pi, \Gamma)$, let M be the maximum length of a string, and let $w = \log |\pi_m^a| + \log |\Delta| + \dim(\Phi) + \dim(\Theta) \geq 2$.*

(i) The growth of $\mathcal{A}$ is bounded as:

$$G(\mathcal{A}, \ell) \leq |\pi_m^a| \cdot |\Delta| \cdot G(\Phi, \ell \cdot M) \cdot G(\Theta, \ell).$$

(ii) The dimension of $\mathcal{A}$ is bounded as:

$$\dim(\mathcal{A}) \leq 2 \cdot w \cdot \log(w \cdot e \cdot M \cdot |\Pi| \cdot |\Gamma|).$$

(iii) The sample complexity of $\mathcal{A}$ is bounded as:

$$S(\mathcal{A}) \in O\big(w \cdot \log(w \cdot M \cdot |\Pi| \cdot |\Gamma|)\big).$$

Next we show the functional description of an automata cascade. It captures the high-level idea that each component of a cascade can be fully executed before executing any of the subsequent automata, since it does not depend on them.

Lemma 2. *The function implemented by an automata cascade $A_1 \ltimes \dots \ltimes A_d$ with $d \geq 2$ can be expressed as*

$$\overline{(I^* \times A_1)} \circ \dots \circ \overline{(I^* \times A_{d-1})} \circ A_d.$$

The cross product with I^* in the functional description above captures the fact that the input of each component is propagated to the next one.

Then, a bound on the growth is derived from the functional description, and hence the dimension and sample complexity bounds.

Theorem 7. *Let $\mathcal{C}$ be a class $\mathcal{A}_1 \ltimes \dots \ltimes \mathcal{A}_d$ where automata classes are $\mathcal{A}_i = \mathcal{A}(\Phi_i, \Delta_i, \Theta_i; a_i, m_i, \Pi_i, \Gamma_i)$, let M be the maximum length of a string, and let $w = \log |\pi_m^a| + \log |\Delta| + \dim(\Phi) + \dim(\Theta) \geq 2$ where dropping the indices denotes the maximum.*

(i) The growth of $\mathcal{C}$ is bounded as:

$$G(\mathcal{C}, \ell) \leq \prod_{i=1}^{d} \left|\pi_{m_i}^{a_i}\right| \cdot |\Delta_i| \cdot G(\Phi_i, \ell \cdot M) \cdot G(\Theta_i, \ell \cdot M).$$

(ii) The dimension of $\mathcal{C}$ is bounded as:

$$\dim(\mathcal{C}) \leq 2 \cdot d \cdot w \cdot \log(d \cdot w \cdot e \cdot M \cdot |\Pi| \cdot |\Gamma|).$$

(iii) The sample complexity of $\mathcal{C}$ is bounded as:

$$S(\mathcal{C}) \in O\big(d \cdot w \cdot \log(d \cdot w \cdot M \cdot |\Pi| \cdot |\Gamma|)\big).$$

By a similar reasoning as in the results for finite classes, the sample complexity has a linear dependency on the number d of cascade components, modulo a logarithmic factor.

Corollary 2. *Let us recall the quantities from Theorem 7, and fix the quantities a_1, m, Φ, Δ, Θ, M, Π, and Γ. Then, the sample complexity of $\mathcal{C}$ is bounded as $S(\mathcal{C}) \in O(d \cdot \log d)$.*

Related Work

The main related work is the sample complexity bounds from (Ishigami and Tani 1997), stated in Theorem 1. Our bounds are qualitatively different, as they allow for describing the sample complexity of a richer variety of classes of automata, and cascades thereof. With reference to Theorem 4, their specific case is obtained when: the input alphabet is non-factored and hence $|\pi_m^a| = 1$; $\Phi = \{I\}$ and hence $|\Pi| = |\Sigma| = k$; Δ is the class of all semiautomata on k letters and n states, and hence $|\Delta| = n^{k \cdot n}$; and Θ is the class of all indicator functions on n states, and hence $|\Theta| = 2^n$. In this case, it is easy to verify that our bound matches theirs.

Learning automata expressed as a cross product is considered in (Moerman 2018). They correspond to cascades where all components read the input, but no component reads the output of the others. The authors provide a so-called *active learning* algorithm, that asks membership and equivalence queries. Although it is a different setting from ours, it is interesting that they observe an exponential gain in some specific cases, compared to ignoring the product structure of the automata.

The idea of decoupling the input alphabet from the core functioning of an automaton is found in *symbolic automata*. The existing results on learning symbolic automata are in the active learning setting (Berg, Jonsson, and Raffelt 2006; Mens and Maler 2015; Maler and Mens 2017; Argyros and D'Antoni 2018).

Conclusion

Given the favourable sample complexity of automata cascades, the next step is to devise learning algorithms, able to learn automata as complex systems consisting of many components implementing specific functionalities.

Acknowledgments

This work has been supported by the ERC Advanced Grant WhiteMech (No. 834228), by the EU ICT-48 2020 project TAILOR (No. 952215), by the PRIN project RIPER (No. 20203FFYLK), by the EU's Horizon 2020 research and innovation programme under grant agreement No. 682588.

References

Abadi, E.; and Brafman, R. I. 2020. Learning and Solving Regular Decision Processes. In *IJCAI*.

Andreas, J.; Klein, D.; and Levine, S. 2017. Modular Multitask Reinforcement Learning with Policy Sketches. In *ICML*.

Angluin, D. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*

Argyros, G.; and D'Antoni, L. 2018. The Learnability of Symbolic Automata. In *CAV*.

Balle, B.; Castro, J.; and Gavaldà, R. 2013. Learning probabilistic automata: A study in state distinguishability. *Theory Comput. Sci.*

Balle, B.; Castro, J.; and Gavaldà, R. 2014. Adaptively learning probabilistic deterministic automata from data streams. *Mach. Learn.*

Berg, T.; Jonsson, B.; and Raffelt, H. 2006. Regular Inference for State Machines with Parameters. In *FASE*.

Brafman, R. I.; and De Giacomo, G. 2019. Regular Decision Processes: A Model for Non-Markovian Domains. In *IJCAI*.

Clark, A.; and Thollard, F. 2004. PAC-learnability of Probabilistic Deterministic Finite State Automata. *J. Mach. Learn. Res.*

De Giacomo, G.; Iocchi, L.; Favorito, M.; and Patrizi, F. 2019. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. In *ICAPS*.

De Giacomo, G.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI*.

Dömösi, P.; and Nehaniv, C. L. 2005. *Algebraic Theory of Automata Networks: An Introduction*. SIAM.

Gaon, M.; and Brafman, R. I. 2020. Reinforcement Learning with Non-Markovian Rewards. In *AAAI*.

Ginzburg, A. 1968. *Algebraic Theory of Automata*. Academic Press.

Haussler, D.; and Long, P. M. 1995. A generalization of Sauer's lemma. *J. Comb. Theory Ser. A*.

Hopcroft, J.; and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Ishigami, Y.; and Tani, S. 1997. VC-dimensions of Finite Automata and Commutative Finite Automata with k Letters and n States. *Discret. Appl. Math.*

Jothimurugan, K.; Bansal, S.; Bastani, O.; and Alur, R. 2021. Compositional Reinforcement Learning from Logical Specifications. In *NeurIPS*.

Kleene, S. C. 1956. Representation of events in nerve nets and finite automata. *Automata studies*.

Krohn, K.; and Rhodes, J. 1965. Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines. *Trans. Am. Math. Soc.*

Maler, O. 1990. *Finite Automata: Infinite Behaviour, Learnability and Decomposition*. Ph.D. thesis, The Weizmann Institute of Science.

Maler, O.; and Mens, I. 2017. A Generic Algorithm for Learning Symbolic Automata from Membership Queries. In *Models, Algorithms, Logics and Tools*.

McNaughton, R.; and Papert, S. A. 1971. *Counter-Free Automata*. The MIT Press.

Mens, I.; and Maler, O. 2015. Learning Regular Languages over Large Ordered Alphabets. *Log. Methods Comput. Sci.*

Moerman, J. 2018. Learning Product Automata. In *ICGI*.

Natarajan, B. K. 1989. On Learning Sets and Functions. *Mach. Learn.*

Neider, D.; Gaglione, J.; Gavran, I.; Topcu, U.; Wu, B.; and Xu, Z. 2021. Advice-Guided Reinforcement Learning in a non-Markovian Environment. In *AAAI*.

Palmer, N.; and Goldberg, P. W. 2007. PAC-learnability of probabilistic deterministic finite state automata in terms of variation distance. *Theory Comput. Sci.*

Ron, D.; Singer, Y.; and Tishby, N. 1996. The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length. *Mach. Learn.*

Ron, D.; Singer, Y.; and Tishby, N. 1998. On the Learnability and Usage of Acyclic Probabilistic Finite Automata. *J. Comput. Syst. Sci.*

Ronca, A.; and De Giacomo, G. 2021. Efficient PAC Reinforcement Learning in Regular Decision Processes. In *IJCAI*.

Ronca, A.; Knorozova, N. A.; and De Giacomo, G. 2022. Automata Cascades: Expressivity and Sample Complexity. *CoRR*, abs/2211.14028.

Ronca, A.; Paludo Licks, G.; and De Giacomo, G. 2022. Markov Abstractions for PAC Reinforcement Learning in Regular Decision Processes. In *IJCAI*.

Schmitt, M. 2004. An Improved VC Dimension Bound for Sparse Polynomials. In *COLT*.

Schützenberger, M. P. 1965. On Finite Monoids Having Only Trivial Subgroups. *Inf. Control.*

Shalev-Shwartz, S.; and Ben-David, S. 2014. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

Toro Icarte, R.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2018. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. In *ICML*.

Vapnik, V. N. 1998. *Statistical learning theory*. Wiley.

Vapnik, V. N.; and Chervonenkis, A. Y. 1971. On The Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory Probab. its Appl.*

Xu, Z.; Gavran, I.; Ahmad, Y.; Majumdar, R.; Neider, D.; Topcu, U.; and Wu, B. 2020. Joint Inference of Reward Machines and Policies for Reinforcement Learning. In *ICAPS*.