Extracting deterministic finite automata from RNNs via hyperplane partitioning and learning

Sandamali Yashodhara Wickramasinghe[1][0000−0002−5550−1940], Jacob M. Howe[1][0000−0001−8013−6941], and Laure Daviaud[2][0000−0002−9220−7118]

[1] City St George's, University of London, UK
{sandamali.wickramasinghe,j.m.howe}@citystgeorges.ac.uk
[2] University of East Anglia, UK
L.Daviaud@uea.ac.uk

Abstract. Recurrent Neural Networks (RNNs) have achieved remarkable success in handling sequential data. However, they lack interpretability. Extracting Deterministic Finite Automata (DFAs) from black-box models can provide insight into their decision-making processes. This research focuses on extracting DFAs from RNNs trained on regular languages using an exact learning framework. The proposed approach employs the L^* algorithm to learn a DFA, and it demonstrates how a hyperplane-based method can be used to partition the RNN state space when evaluating equivalence queries.

Keywords: Recurrent Neural Networks · Finite State Automata · Explainable AI.

1 Introduction

This research explores the extraction of automata from Recurrent Neural Networks (RNNs). Recurrent Neural Networks are known for successfully handling sequential data, making them widely used for language modelling tasks. Unlike feed-forward neural networks, RNNs' sequential processing architecture allows them to capture the patterns and dependencies of language data.

Regardless of their effectiveness, owing to their lack of interpretability, RNNs are considered to function as black boxes [2,15,22]. The internal decision-making process of RNNs is difficult to understand, and there is no straightforward way to interpret it or the network output at each step. This lack of transparency raises questions about the model training process and whether the model has captured the specifics of the problem it is supposed to address. This is not just a theoretical issue and has practical consequences, specifically in cases where understanding the decision-making process is important [4,14].

To address this, researchers have been exploring methods to extract more interpretable models, such as automata, to approximate the functionality and behaviour of RNNs. Automata are symbolic, rule-based models that provide a clearer structure, making it easier to understand and verify the model's behaviour. While they can be used to interpret the RNN's decision-making process, they can also serve as a substitute for the RNN when inferring.

One of the first attempts to extract an automaton from RNNs was presented in [7], where it was demonstrated that the structure of the state space of an RNN trained on a regular grammar could be closely approximated by a corresponding minimal finite-state automaton. The work in [11] further explored this connection by introducing a quantisation algorithm that divides the RNN state space into partitions for automaton extraction. Similarly, [23] showed that clustering the output space of an RNN is an effective method for extracting finite-state representations. Furthermore, [33] provided evidence that the hidden state space of an RNN forms cluster-like structures while learning a regular grammar. Also, [29] use k-means clustering to extract a Deterministic Finite Automata (DFA), but this requires predefining the number of DFA states as a hyperparameter [29].

The L^* *algorithm* [1] is a widely recognised *exact learning* method used to learn a minimal DFA that accepts the same language as an unknown target system. Exact learning is a framework where a learner interacts with an oracle (or teacher) to identify the target model. The goal of the learner is to construct a hypothesis, such as a DFA, that matches the behaviour of the target system, using a series of carefully chosen queries [12]. The L^* algorithm uses membership queries and equivalence queries to learn the DFA, where membership queries ask whether a specific string belongs to the target language, whilst equivalence queries check if a proposed automaton matches the target language and return a counterexample if it does not.

To apply L^* with RNNs as the oracle, answers to membership and equivalence queries are needed. Membership queries can be straightforwardly answered, but since RNNs are defined over high-dimensional continuous vector spaces, whereas DFAs have a finite set of discrete states, a notion of equivalence is not immediately available. Building on earlier work, [32] explores the extraction of DFAs from RNNs using the L^* algorithm. Their approach introduces an abstraction of the RNN state space for the equivalence query. This abstraction was constructed by partitioning the RNN hidden state space using support vector machine (SVM) classifiers trained on the hidden states, a potentially expensive operation. By iteratively partitioning the RNN state space only when necessary, they demonstrate that it is possible to extract a DFA from an RNN while mitigating state explosion.

In this work, an alternative DFA extraction method using the L^* algorithm is presented, where the equivalence queries are realised using a hyperplane-based partitioning of the RNN state space. This abstraction does not construct a DFA itself but serves as an intermediate representation that allows comparison of the behaviour of the RNN with the hypothesis DFA extracted by L^*. The main contributions include a hyperplane-based approach to partitioning the RNN state space, which is computationally attractive, and the way that this is integrated into a method for answering equivalence queries. In addition, using k-means clustering to start with an initial number of clusters, based on the natural clustering of RNN hidden states, in order to reduce the number of refinements needed in partitioning, is investigated. An experimental evaluation of the work on formal language problems is presented.

The remainder of this paper is organised as follows. Sections 2 and 3 review related work and provide background information and definitions relevant to the study. Section 4 describes the proposed methodology, focusing on how the L^* algorithm is employed for automata extraction. Section 5 presents the experimental setup and results, followed by a discussion of the findings in Section 6 and conclusions in Section 7.

2 Related work

Neural nets were first introduced in [20], with a model inspired by biological neurons. Foundational concepts involving the use of rational expressions and finite-state automata to describe the expressive power of such models were later established in [16]. Extending these foundational concepts, [27] explored the computational capabilities of first-order RNNs, where they argue that RNNs can simulate Turing machines under specific configurations, thereby establishing a theoretical framework that highlights the computational potential of neural networks. Building on this, recent research has focused on the extraction of automata from trained RNN models, which shows that there is a possibility to extract finite automata from RNNs trained for grammatical inferences [11,15,23].

The automata extraction methods can be divided into two main categories based on their underlying principles: compositional approaches and learning approaches [15]. Compositional approaches analyse and partition the RNN state space to identify representative state vectors and the transitions between them, typically using clustering to capture the discrete dynamics of the RNN. Learning approaches, such as exact learning, build an automaton by querying the target system's outputs and iteratively refining the model using counterexamples.

In the early days following the introduction of RNNs [9], various studies explored the possibility of obtaining finite automata by partitioning the state space of RNNs [7,11,23,33], which laid the foundation for automata extraction. An early attempt to verify the structure of the RNN state space using clustering algorithms, specifically hierarchical clustering, was presented in [7]. This work aimed to closely approximate the corresponding minimal finite-state automaton by organising the states of the RNN into meaningful clusters. This approach provided an early demonstration of how unsupervised learning techniques could be applied to uncover the underlying state transitions of RNNs. Building on this work, further research in [11] explored the use of state space partitioning through clustering, highlighting its importance in understanding and modelling the behaviour of RNNs.

The study in [23] continued this line of inquiry by employing a simple clustering algorithm to partition the RNN state space, using input data generated from predefined DFAs. Their work used a quantisation method, which converts continuous internal states of RNNs into discrete representations. Here, an initial quantisation level has to be fixed before extraction. Their results indicated that a minimal DFA could be effectively extracted through the method by grouping like-valued state vectors together as a single cluster and treating a cluster as a

state of a DFA. There are subsequent studies which have expanded and refined these approaches for automata extraction. For instance, one can use a clustering method such as k-means [33,29], which does not need quantisation [29]. Quantisation can lead to a state explosion where the number of discrete states grows exponentially, making the extraction process more complicated.

The primary advantage of compositional approaches lies in their scalability, as these methods can process extensive datasets and derive automata representations without manual intervention. It makes them particularly suitable for applications where the RNN operates over a wide range of inputs and generates complex output patterns [15,29]. However, these methods also face significant challenges, such as ensuring the accuracy of the state space partitioning, as improper clustering can lead to the loss of important information and result in an incomplete or inaccurate automaton [8].

Learning approaches use exact learning algorithms, such as the L^* algorithm [1], to extract an automaton from a target system. Here, an automaton is extracted by asking queries to understand the underlying state transitions of a network. This approach has achieved wide success, specifically for regular languages. However, unlike compositional methods, such approaches have limitations in scalability [30].

A prominent example of work in this area is the research presented in [32], which employs the L^* algorithm originally developed in [1]. The L^* algorithm is a well-established exact learning method that uses an oracle to query a target network and incrementally construct an equivalent automaton. Variants of this algorithm have also been applied in other studies focused on automaton extraction [19].

The work in [32] adapted the L^* algorithm to extract a DFA from RNNs trained with perfect train and test accuracy. Their novel algorithm efficiently utilises the L^* algorithm and an abstraction of the state space of the RNN in extracting automata [32]. Later, this line of work was extended to extracting deterministic weighted automata using a variation of the L^* algorithm adapted to a probabilistic setting. However, this involved performing equivalence queries using the oracle itself [31]. Additionally, several studies have explored the extraction of weighted finite automata using variants of the L^* algorithm [10,21], originally proposed in [3].

3 Background

This section introduces the technical notation that will be used throughout the paper. See [26] for details.

3.1 Languages and automata

Formal grammars are used to model and to study the properties, structures, and behaviours of computational systems from the Chomsky hierarchy [5]. Examples

of common formal grammars include regular grammars, context-free grammars, and recursively enumerable grammars.

A finite alphabet Σ is a finite set of elements, called letters and the set of all finite words (a.k.a. strings, i.e. a finite sequence of letters) over Σ is denoted by Σ^*. The empty word of length 0 is denoted by ε. The length of a word $w \in \Sigma^*$ is denoted by $|w|$. Given two words u and v, their concatenation is the word $w = uv$, and u is said to be a prefix, and v a suffix of w, respectively.

Definition 1. *A* Deterministic Finite Automata (DFA) *A is a tuple $\langle \Sigma, Q, q_I, F, \delta \rangle$, where Σ is the finite alphabet, Q is the finite set of states, $F \subseteq Q$ are the accepting states, $q_I \in Q$ is an initial state and $\delta : Q \times \Sigma \to Q$ is the transition function.*

A word $w = \sigma_1\sigma_2 \dots \sigma_n \in \Sigma^*$, with $\sigma_i \in \Sigma$ for all i, is *accepted* by A, if there exists $q_0, q_1, q_2, \dots, q_n \in Q$, such that $q_0 = q_I, q_n \in F$ and $\delta(q_{i-1}, \sigma_i) = q_i$. The regular language recognised by A, is denoted by L_A, which is defined by the set of words accepted by the DFA A. Two DFAs are called equivalent if they recognise the same language.

3.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks that includes at least one feedback loop, allowing activations to circulate within the network. This recurrent structure makes RNNs particularly well-suited for modelling sequential data. By utilising their internal state, RNNs can retain memory of previous elements in a sequence, enabling them to consider the impact of past inputs when computing current outputs. Thus, the units in an RNN layer use not only the input from the previous layer but also the hidden state from the previous time step. The output at a given time step becomes part of the input for the next, facilitating the sequential mechanism within the RNN. Later advancements, such as Long Short-Term Memory (LSTM) [13] (and gated Recurrent Unit (GRU) [6]) have significantly improved RNNs' ability to handle long-range data dependencies, thus expanding their potential uses. In this work, the RNNs with LSTM and GRU architectures will be discussed.

An RNN that processes an input word $w \in \Sigma^*$ produces an output and can also provide access to the corresponding hidden state information. Formally [21,32], a *Recurrent Neural Network (RNN)* is a tuple $R = (h_0, f_R, g_R)$, where $h_0 \in \mathbb{R}^d$ is the initial state, where $d \in \mathbb{N}$ is the dimension of the RNN state space, $g_R : \mathbb{R}^d \times \Sigma \to \mathbb{R}^d$ is the transition function producing the next hidden state given the current hidden state and an input symbol, $f_R : \mathbb{R}^d \to \{0, 1\}$ is the output function. Here, the RNN R is a binary acceptor.

For any $w \in \Sigma^*$ the transition function g_R can be extended by defining the function $g_R^* : \mathbb{R}^d \times \Sigma^* \to \mathbb{R}^d$ as follows:

$$g_R^*(h, \varepsilon) = h, \quad g_R^*(h, w\sigma) = g_R(g_R^*(h, w), \sigma)$$

for $h \in \mathbb{R}^d$, $w \in \Sigma^*$, and $\sigma \in \Sigma$.

The hidden state of the RNN R after processing a word $w \in \Sigma^*$ is $g_R^*(h_0, w)$. The output of the RNN on input word w is then $f_R(g_R^*(h_0, w)) \in \{0, 1\}$. The set of all possible hidden states reachable from h_0 is denoted by $S \subseteq \mathbb{R}^d$.

3.3 L^* Algorithm for DFA learning.

Angluin's L^* algorithm [1] is an exact learning algorithm designed to learn a minimal DFA for regular languages through a series of *membership and equivalence queries*. It interacts with an oracle, known as a *minimally adequate teacher*, which is a theoretical construct capable of answering these two types of queries about an unknown regular language L over an alphabet Σ.

A membership query provides a word w from Σ^*, and the oracle classifies the word as either belonging to the language (`true`) or not (`false`). Based on these responses, the learner constructs a DFA A_L that hypothesises acceptance of the language L (see Definition 2). Equivalence queries are then used to verify whether the learned DFA A_L correctly accepts the language L. If A_L does not accept L, the oracle provides a counterexample. The learner uses this feedback to refine the DFA.

The algorithm iteratively refines the DFA through further membership queries until the oracle confirms that A_L accepts the language L. At this point, the algorithm terminates, and A_L is returned as the final DFA. In the L^* algorithm, an observation table is maintained to organise the results of membership queries. The table is incrementally filled by querying the oracle during the learning process.

The *observation table* T in the L^* algorithm consists of a matrix with rows indexed by a finite set of prefixes U and columns by a finite set of suffixes V. The entry at position (u, v) is 1 if $uv \in L$, and 0 otherwise, where $u \in U$ and $v \in V$. From an observation table (initially set with $U = V = \{\varepsilon\}$), the L^* algorithm performs membership queries to fill the table until it is *closed*.

The *closed condition* states that for every $u\sigma$, where u indexes a row and $\sigma \in \Sigma$, there exists a word u' indexing a row of T that is identical to what the row indexed by $u\sigma$ would be, i.e. the role of an extended word $u\sigma$, using $\sigma \in \Sigma$, is already covered by some existing row in T.

The membership queries are performed for all concatenations of prefixes in U with suffixes in V. When the table is not *closed*, U is updated by adding new prefixes that correspond to uncovered transitions, and when the table is not *consistent*, V is extended by adding distinguishing suffixes. The *consistency condition* states that for any two prefixes $s_1, s_2 \in U$, where U is the set of prefixes indexing the rows of the observation table, if the rows indexed by s_1 and s_2 are identical, for all $\sigma \in \Sigma$, rows indexed by $s_1\sigma$ and $s_2\sigma$ are equal.

When the observation table T is *closed* and *consistent*, the algorithm returns a DFA A_L, as defined in Definition 2, to consider for the equivalence queries. The equivalence queries verify whether the DFA A_L correctly recognises the target language. If it does not, a counterexample $w \in \Sigma^*$ is returned to the learner L^*. Then, there exist strings u and v such that $w = uv$, and by updating the observation table to include u in U (the set of prefixes) and v in V (the set of

suffixes), the table remains consistent. Then the table T is extended by adding rows $u\sigma$ by using the membership queries for the missing entries where $\sigma \in \Sigma$.

Definition 2. *Let Σ be a finite alphabet. Given a closed and consistent observation table defined by two finite sets of words U and V and $T : U \times V \to \{0, 1\}$ (such that $T(u, v) = T(u', v')$ if $uv = u'v'$), the* hypothesis DFA *is defined by $A = (\Sigma, Q, q_I, F, \delta)$ where Q is the finite set of distinct rows indexed by words in U, $q_I \in Q$ is the row indexed by ε, $F \subseteq Q$ is the rows indexed by $u \in U$ such that $T(u, \varepsilon) = 1$, and $\delta : Q \times \Sigma \to Q$ is the transition function such that for a state q corresponding to a row indexed by a word u, and for a letter $\sigma \in \Sigma$, $\delta(q, \sigma)$ is the state corresponding to the row indexed by the prefix $u\sigma$. Note that δ is well defined, since the table is closed and consistent.*

4 Extracting a DFA from an RNN via learning

This section describes the procedure for extracting a DFA from an RNN.

The DFA is extracted using the L^* algorithm, as explained in Section 3.3. Here, the oracle is the RNN R, and the learner L^* attempts to infer the language recognised by R. During the membership queries, words are queried through the RNN R, and the outputs are recorded in an observation table with $f_R(uv) \in \{0, 1\}$. Once the observation table T is closed, the algorithm returns a DFA to be evaluated through equivalence queries.

Membership queries pose no issue; the main challenge in using the L^* algorithm lies in answering equivalence queries. RNNs operate over high-dimensional continuous vector spaces, whereas DFAs consist of a finite set of discrete states. Due to the continuous and potentially infinite nature of RNN state representations, checking formal equivalence between an RNN and a DFA is undecidable in general [18]. One way to address this is by using an abstraction of the RNN's hidden state space – typically by clustering hidden states to induce a finite-state model [32]. This abstraction allows for the use of equivalence queries, in which the abstracted automaton is iteratively compared against the L^* DFA. Counterexamples identified through these queries are used to refine the DFA until convergence is reached or equivalence is reasonably approximated. In this work, this challenge is addressed using an alternative approach.

4.1 Partitioning the state space of the RNN

The state space of an RNN can be partitioned based on the behaviour of its hidden states. During sequence processing, hidden states that exhibit similar dynamics tend to lie close to one another in the high-dimensional space. These hidden states can therefore be grouped into clusters, which reveal structural patterns in the RNN's internal representation. This abstraction can be used to perform equivalence queries and serve as the basis for constructing the L^* hypothesis DFA. Unlike in the other works, here a partitioning of the RNN state space is maintained *rather than* an exact DFA.

Given a recurrent neural network (RNN) R with a state space $S \subseteq \mathbb{R}^d$, a partitioning $P = \{p_i \subseteq \mathbb{R}^d | i, j \in \{0, \ldots, n\}, p_i \cap p_j = \emptyset, \cup_{i=0}^{n} p_i = \mathbb{R}^d\}$ divides space into non-overlapping regions. The abstraction A_P is the hypothesised abstraction of the RNN state space, represented as the collection of all partitions with each partition p_i mapped to 1 or 0, based on the classification of the first hidden state encountered within that partition.

To extract an abstract partitioning A_P, the state space of the RNN R can be explored according to the network's transition function g_R. This exploration begins with an initial partition $p(h_0)$, where h_0 represents the initial hidden state of the RNN. Initially, the entire state space is treated as a single partition. As exploration progresses through equivalence queries, this space is incrementally divided into smaller partitions. The classification of a partition is determined based on the first hidden state observed within it, during the processing of an input word. If the next hidden state, corresponding to a subsequent input symbol in the word, falls within the same partition, a loop transition is created. Otherwise, the transition leads to a different partition. The partitions are stored in a decision tree structure.

4.2 Identifying conflicts

The equivalence between the L^* DFA A_L and the abstraction A_P should produce the same outputs for the same inputs, meaning they recognise the same language or accept the same set of input-output behaviours.

The equivalence query has two tasks: it should be able to return a counterexample in the event where the L^* DFA is incorrect, and it should be able to refine the partitioning P, if A_P is incorrect. As presented in [32], exploring the DFA and the partitioning in parallel allows for inconsistencies between them to be identified without fully exploring the abstraction partitioning A_P.

In equivalence queries, the extracted DFA A_L is compared against the current partitioning of the RNN state space, denoted by A_P. Each word $w \in \Sigma^*$, up to the maximum length of the training sequences, is submitted to both A_L and A_P. The states reached and their corresponding classifications are then compared to detect any discrepancies.

If the classifications of the reached states differ for a given input w, the input is passed to the RNN R to determine the source of the discrepancy. If A_L is incorrect according to R, i.e., $A_L(w) \neq f_R(w)$, the mismatch is referred to as a *conflict in the DFA*. Additionally, at each transition, the classification of the hidden states in R is compared with that of A_L to identify cases where both A_L and A_P may fail to represent the correct behaviour. If disagreement is found, these are also treated as *conflicts in the DFA*.

A *partition conflict* is identified when A_P behaves inconsistently with R, specifically when $A_P(w) \neq f_R(w)$. This indicates that a single partition in A_P maps to multiple states in the minimal DFA A_L. Such a one-to-many mapping is invalid, as each state in A_L must correspond to a unique behaviour.

Additionally, during the equivalence queries, each hidden state explored is checked against the current partition to ensure that no conflicting hidden states

are grouped within the same partition. Specifically, this refers to a scenario where both a positively and a negatively classified hidden state (i.e., one leading to acceptance and one to rejection) reside within the same partition. Algorithm 1 details the procedure for performing the equivalence queries during the extraction.

Algorithm 1 Equivalence Queries

Require: RNN $R = (h_0, f_R, g_R)$, learning DFA A_L, abstract partitioning A_P, `visited` $\leftarrow \emptyset$
Ensure: Counterexample, Refinement, or Equivalence
for each input word $w \in \Sigma^*$ **do**
  Add w to `visited`
  Compute $h \leftarrow g_R^*(h_0, w)$
  if $A_L(w) \neq A_P(w)$ and $A_L(w) \neq f_R(w)$ **then**
    return w as a `Counterexample`
  else if $A_L(w) \neq A_P(w)$ and $A_P(w) \neq f_R(w)$ **then**
    Let $w_1 \leftarrow w$
    for each $w_2 \in$ `visited` $\setminus \{w_1\}$ **do**
      if there exists a suffix s such that $A_L(w_1 s) \neq A_L(w_2 s)$ and $A_P(w_1 s) = A_P(w_2 s)$ **then**
        Let $h_1 \leftarrow g_R^*(h_0, w_1 s)$, $h_2 \leftarrow g_R^*(h_0, w_2 s)$
        `Refinement`(h_1, h_2)
        return refined A_P
      end if
    end for
  else if there exists a hidden state h' such that $f_R(h') \neq A_P(h')$ **then**
    conflicting h' in the same p_i, where $i \in \{0, 1, 2, ...\}$
    Let h_μ be the mean of all $h \in p_i \setminus \{h'\}$
    `Refinement`(h', h_μ)
    return A_P
  end if
end for
return `Equivalence`

4.3 Resolving conflicts

There are two possible ways to resolve conflicts. If there are *conflicts in the DFA* A_L, the conflicting word w is returned to the learner L^* as a counterexample. The observation table T is then updated with the appropriate prefix and suffix of the counterexample, leading to a refined L^* DFA A_L.

If there is a *partition conflict*, it indicates an inconsistency in the partitioning of the RNN state space. Specifically, there exist two input sequences, w_1 and w_2, that follow distinct paths in A_L but are mapped to the same path in A_P. This means that two different states in A_L are mapped into a single partition in A_P, resulting in a conflict.

To resolve this, the conflicting partition is split using the hidden states corresponding to w_1 and w_2, leading to a partition refinement. These hidden states are selected by identifying sequences of equal length that behave differently in

the L^* DFA but are assigned to the same abstract partition in A_P. This discrepancy reveals that the current partitioning fails to distinguish between sequences that should lead to different outcomes. By refining A_P—i.e., separating the conflicting hidden states into distinct partitions—the abstraction is updated so that w_1 and w_2 are no longer assigned to the same partitions.

Additionally, there can be partition conflicts due to conflicting hidden states that are grouped within the same partition. The refinement process is done by computing the mean of all the existing hidden states in the current partition and comparing it with the conflicting hidden state. These two representatives—the partition mean and the conflicting hidden state—are then passed to a refinement operation to split the conflicted partition, thereby creating more precise partitions that align with the actual behaviour of the RNN. Refinements are applied only when conflicts are detected within a partition.

Refinements were done using a hyperplane-based approach as described below.

Using a hyperplane for the partition refinement. A hyperplane is a decision boundary that separates a high-dimensional space into two distinct regions. In the context of partition refinement, a hyperplane is used to divide a conflicted partition into two, ensuring that the points causing the conflict are assigned to distinct partitions, maximising the distance between them for better separation.

Suppose w_1 and w_2 are conflicting words identified under partition conflicts. Then, the hidden states $h_1 = g^*(h_0, w_1)$ and $h_2 = g^*(h_0, w_2)$, which exist in the conflicted partition, should be passed into a refinement operation, so that they are mapped into two separate partitions.

A hyperplane can be defined as $z.x + b = 0$, where $z \in \mathbb{R}^d$ is the normal vector to the hyperplane, $x \in \mathbb{R}^d$ is a point in the d-dimensional space, $b \in \mathbb{R}$ is the bias, determining the hyperplane's position relative to the origin. Hidden states $h_1, h_2 \in \mathbb{R}^d$ of the RNN are high-dimensional vectors. The hyperplane separating the two hidden states can be calculated as follows.

$$z = h_2 - h_1 \qquad m = \frac{h_1 + h_2}{2} \qquad b = -z \cdot m \qquad z \cdot x + b = 0$$

For a given hidden state h, the signed distance from the hyperplane is $d(h) = z \cdot h + b$. Thus, the partition refinement function is,

$$P(h) = \begin{cases} p_i, & \text{if } d(h) > 0, \\ p_{i+1}, & \text{if } d(h) \leq 0. \end{cases}$$

where, $i = 1, 2, 3...$ is the partition index.

The hyperplane passes through the midpoint of the two conflicting hidden states and is perpendicular to the line joining these vectors. This hyperplane divides the d-dimensional state space into two distinct partitions, ensuring h_1 and h_2 belong to separate partitions. This condition is checked before accepting the partition refinement.

During each iteration of the refinement process, a conflicted partition is divided into two, resulting in an incremental increase in the total number of partitions, i.e. abstract partitions in A_P. A decision tree structure was employed to manage and store these partitions. Each hyperplane that divides a partition is represented as a leaf node in the decision tree, with the resulting partitions as its child nodes.

This hierarchical structure allows the decision tree to dynamically expand with each new round of partition refinement, accommodating additional hyperplanes as they are introduced. The decision tree facilitates tracking transitions when a word w is fed into the abstract partitioning A_P. For each word w, the corresponding hidden states at each time step are obtained and then passed through the decision tree to determine their respective partitions.

The procedure is described in Algorithm 2, which elaborates the refinement operation mentioned in line 11 of Algorithm 1.

Algorithm 2 Partition Refinement using Hyperplane

Require: $R, A_P = \{p_i \mid i \in \{0, 1, 2, \ldots, n\}\}$, conflicted partition : p', decision tree : D, conflicted hidden states : h_1, h_2
Ensure: Refined A_P, updated : D
1: Define hyperplane: $z \leftarrow h_2 - h_1$, $b \leftarrow -z \cdot \frac{h_1+h_2}{2}$
2: **for** each h in p' **do**
3: Compute signed distance $d(h) \leftarrow z \cdot h + b$
4: Assign h to p_i if $d(h) > 0$, otherwise to p_{i+1}
5: **end for**
6: Store hyperplane $z \cdot x + b = 0$ at the conflicted node of D
7: Add p_i and p_{i+1} as child nodes of D
8: **return** `Refined` A_P, `updated` : D
9: **Termination**: Proceed to the next iteration of Algorithm 1

4.4 Initial partitioning with *k*-means clustering

Initially, the RNN state space was treated as a single partition, and refinements were applied only when necessary. To potentially improve efficiency, an initial partitioning using k-means clustering is introduced. The hidden states of the RNN exhibit a natural clustering structure due to variations in their weight-based representations. This property was leveraged to divide the state space into multiple initial partitions, allowing the extraction process to begin with a more informative abstraction. As a result, fewer refinements were required later, reducing the overall effort needed for exploration and partition refinements.

k-means is a clustering algorithm that groups data points based on the Euclidean distance metric. The proposed method begins the extraction process with k clusters—that is, k abstract partitions. To determine an appropriate value for k, the elbow method was employed, which estimates the natural number of clusters present in the hidden state space [17].

4.5 Termination of the Equivalence queries

The algorithm's termination strategy was adapted based on specific observations. In most cases, the extraction process completes within seconds. However, there were instances where the extraction algorithm required significantly more time due to having to evaluate the RNN model while doing the equivalence queries. To address such cases and prevent the algorithm from running indefinitely, terminating conditions were introduced.

Initially, a termination condition based on the length of a sequence was introduced. The counterexample search proceeds in lexicographical order. When a counterexample is identified, the equivalence check is repeated to verify whether the refined DFA A_L is equivalent to the RNN. If the equivalence is not satisfied, the process either provides a new counterexample or refines the partitioning A_P. If no counterexample is found and the sequence length exceeds the previous counterexample's length by more than five characters, the algorithm terminates.

The partition refinement process is performed to obtain an abstract partition A_P that can be explored in parallel to help identify counterexamples during the equivalence queries. However, an excessive number of refinements without yielding any counterexamples is inefficient and unproductive. Therefore, the algorithm is designed to terminate if more than 10 refinements are made without finding any counterexamples.

The algorithm is designed to operate until it can no longer identify any new counterexamples within the predefined word length limits. If the extraction process takes excessive time, a time limit is invoked. The set limit is 10 seconds. If no additional counterexamples are identified after exhausting all possibilities within that time, the algorithm terminates and outputs the final L^* DFA A_L, which is accepted as the final DFA.

5 Experiments and Results

5.1 Model training

The implementation is in Python, using PyTorch for neural network construction and scikit-learn for additional machine learning functions [24,25]. The target languages are infinite, so the training data is a sample from the language: for languages with an alphabet of 2 elements, sequences ranging from lengths 1 to 15 were used; if the alphabet contains more than 2 elements, the maximum sequence length was limited to 10. Each string was labelled as $true$ or $false$ depending on whether it belonged to the target language.

For each language, the sample data was initially divided into training (80%), validation (10%), and test sets (10%), with each selection reflecting the proportion of positive and negative samples in the whole dataset. Many of the languages are highly imbalanced; therefore, oversampling was used in the training set to ensure a balanced representation of labels, thereby mitigating imbalance and improving the model's ability to generalise.

The models were trained using one hidden layer with 16 hidden states, the Adam optimiser, and an initial learning rate of 0.0001. Each RNN was trained for up to 50 epochs on the training set, with an early stopping rule implemented, halting training if the validation accuracy did not improve for more than 5 consecutive epochs. If the desired accuracy was not achieved, training was extended up to 100 epochs. All models aimed to achieve 100% training and testing accuracy, with some achieving this benchmark, demonstrating perfect learning of the training data. However, the training process did not always result in perfect accuracy. The primary objective of this work is learning DFAs from RNNs, not training RNNs. Therefore, for some languages, the DFA extraction was performed on RNNs that have not completely captured the input language.

5.2 Datasets

Tomita Grammars The Tomita grammars [28] are a set of seven regular grammars over the binary alphabet $\Sigma = \{0, 1\}$. These grammars are widely used as benchmarks for evaluating automaton learning, grammar inference, and explainable AI methods. Each grammar defines a distinct language with varying levels of complexity, making them valuable for testing the capabilities of models such as RNNs and automaton extraction techniques.

Further Regular languages Further sample regular languages were also used to test the algorithm. These languages primarily consisted of sequences over the binary alphabet $\Sigma = \{0, 1\}$ and alphabet of size 3: $\Sigma = \{a, b, c\}$. However, the algorithm is not limited to these and can be applied to RNNs trained on regular languages over any arbitrary alphabet. The 7 Tomita grammars and 20 regular languages are given below:

T1: 1^*	RL1: string start with 1	RL11: string without 11
T2: $(10)^*$	RL2: string start with 0	RL12: string contains 11
T3: no odd run of 1s follows an odd run of 0s	RL3: string end with 0	RL13: len ≥ 5, 5th from right 1
	RL4: string end with 1	RL14: alternate $0, 1$
T4: no substring 000	RL5: string contains 101	RL15: every 0 follow by 1, end 1
T5: even number of both 0s and 1s	RL6: string contains 0101	RL16: a^*b or c^*b
	RL7: string start with 0, end 1	RL17: $(aa)^*(bbb)^*$
T6: $\#0 - \#1 =$ multiple of 3	RL8: len$\geq$ 3, 3rd symbol 0	RL18: bc^*b or ac^*a
T7: $0^*1^*0^*1^*$	RL9: 0^*1^*01	RL19: 0^n1^m, $n \geq 3$, m even
	RL10: string contains 00	RL20: string with at most two 0s

5.3 Extracting DFA from LSTMs and GRUs

RNN models with LSTM and GRU architecture, for the regular language datasets, were trained as described in Section 5.1. The DFA extraction algorithm was then run on the resulting RNNs with a timeout of 10 seconds. Tables 1 and 2 collate the results. The dataset used for testing includes the full set of sequences generated up to the maximum sequence length, 15 when $|\Sigma| = 2$, and 10 when $|\Sigma| = 3$. Additionally, an experiment was conducted to compare the extraction time when SVMs were used instead of hyperplanes in the partitioning method.

Table 1. Performance of automata extracted from LSTMs. Columns represents: target language L, performance of LSTM R compared to L, performance of the extracted DFA A_L compared to R, performance of A_L compared to L, number of partitions in A_P, number of states $|Q_A|$ in A_L, total time t for the extraction (seconds) using hyperplanes, and the time $ref.t$ of the final refinement/counterexample (seconds) using the hyperplanes, total time t for the extraction (seconds) using the SVM method, and the time $ref.t$ of the final refinement/counterexample (seconds) using the SVM method. Each row represents languages for which the LSTM models were trained.

L	R *vs* L			A_L *vs* R			A_L *vs* L			$\|P\|$	$\|Q_A\|$	*Hyperplane*		*SVM*	
	Acc.	*Prec.*	*F1*	*Acc.*	*Prec.*	*F1*	*Acc.*	*Prec.*	*F1*			$t(s)$	$ref.t(s)$	$t(s)$	$ref.t(s)$
T1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	2	0.23	0.08	2.18	0.13
T2	0.99	1.00	0.93	1.00	1.00	1.00	0.99	1.00	0.93	9	4	10.0	0.48	10.00	0.98
T3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	5	5	3.39	1.42	8.09	1.56
T4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	6	4	10.00	1.79	10.00	2.61
T5	0.83	0.50	0.67	1.00	1.00	1.00	0.83	0.50	0.67	7	4	10.00	1.31	10.00	3.5
T6	0.99	1.00	0.99	1.00	1.00	1.00	0.99	1.00	0.99	11	4	2.47	2.46	10.00	2.87
T7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	12	5	10.00	4.24	10.00	4.07
RL1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	3	0.63	0.15	10.00	0.19
RL2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	3	0.64	0.20	10.00	0.53
RL3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	2	1.10	0.10	2.01	0.17
RL4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	2	0.22	0.10	1.59	0.16
RL5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	12	4	10.00	1.74	10.00	6.67
RL6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	11	5	10.00	6.64	10.00	9.73
RL7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	7	4	10.00	0.81	10.00	0.90
RL8	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8	5	10.00	5.85	10.00	0.44
RL9	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	10	7	10.00	0.79	10.00	1.74
RL10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	7	3	10.00	0.78	10.00	1.57
RL11	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	3	0.66	0.16	10.00	0.21
RL12	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	6	3	10.00	0.74	10.00	2.00
RL13	0.99	0.99	0.99	0.68	0.75	0.64	0.68	0.75	0.64	17	28	10.00	9.98	10.00	9.99
RL14	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	5	4	3.08	0.43	6.92	0.52
RL15	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4	3	10.00	0.41	10.00	0.69
RL16	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8	5	10.00	3.47	10.00	8.82
RL17	0.99	1.00	0.98	1.00	1.00	1.00	0.99	1.00	0.98	10	7	10.00	0.98	10.00	2.17
RL18	0.99	0.80	0.89	1.00	1.00	1.00	0.99	0.80	0.89	16	11	10.00	8.86	10.00	6.29
RL19	0.93	0.01	0.02	0.94	0.94	0.19	0.99	0.09	0.16	14	13	10.00	9.79	10.00	9.98
RL20	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	9	4	10.00	4.25	10.00	2.62

The final set of experiments investigates the impact of initialising the DFA extraction algorithm with an initial partitioning determined by performing k-means clustering on the weight space of the trained RNN. Table 3 gives the result of repeating the DFA extraction from the LSTMs with this pre-processing step.

6 Evaluation and Discussion

The results show that the algorithm effectively extracts DFAs from both LSTM and GRU models. Among the 27 LSTMs, 20 achieved 100% training accuracy, and five reached 99%. However, among these, three models (T2, RL17, RL18) exhibited low F1-scores despite high accuracy, suggesting that they failed to capture the underlying language. Two models (T5, RL19) showed both low accuracy and F1-scores, with RL19 performing particularly poorly (F1-score 2%). DFA extraction was successful in 25 out of 27 cases, achieving 100% accuracy. The exceptions, RL13 and RL19, terminated due to time limits, producing DFAs

Table 2. Performance of automata extracted from GRUs. Columns represents: target language L, performance of the GRU R compared to L, performance of the extracted DFA A_L compared to R, performance of A_L compared to L, number of partitions in A_P, number of states $|Q_A|$ in A_L, total time t for the extraction (seconds) using hyperplanes, the time $ref.t$ of the final refinement or counterexample (seconds) using the hyperplanes, total time t for the extraction (seconds) using the SVM method, and the time $ref.t$ of the final refinement or counterexample (seconds) using the SVM method. Each row represents languages for which the GRU models were trained.

L	R *vs* L			A_L *vs* R			A_L *vs* L			$\|P\|$	$\|Q_A\|$	*Hyperplane*		*SVM*	
	Acc.	*Prec.*	*F1*	*Acc.*	*Prec.*	*F1*	*Acc.*	*Prec.*	*F1*			$t(s)$	$ref.t(s)$	$t(s)$	$ref.t(s)$
T1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	2	0.23	0.09	0.57	0.13
T2	0.99	1.00	0.93	1.00	1.00	1.00	0.99	1.00	0.93	8	4	10.00	0.57	10.00	0.98
T3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8	5	10.00	2.44	10.00	2.96
T4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8	4	10.00	1.74	10.00	3.57
T5	0.83	0.50	0.67	1.00	1.00	1.00	0.83	0.50	0.67	7	4	10.00	4.87	10.00	3.47
T6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	5	3	10.00	0.17	10.00	0.25
T7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8	5	10.00	8.79	10.00	8.90
RL1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	3	0.61	0.14	10.00	0.20
RL2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	3	0.63	0.13	10.00	0.17
RL3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	2	0.22	0.09	0.33	0.11
RL4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	2	0.24	0.09	0.32	0.14
RL5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	10	4	10.00	7.61	10.00	4.26
RL6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8	5	10.00	7.51	10.00	7.28
RL7	0.99	0.98	0.99	0.99	1.00	0.99	0.99	0.99	0.99	11	9	10.00	2.08	10.00	3.91
RL8	0.99	0.99	0.99	1.00	1.00	1.00	0.99	0.99	0.99	14	7	10.00	2.87	10.00	6.79
RL9	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	14	7	10.00	6.43	10.00	4.87
RL10	0.95	1.00	0.97	0.95	0.95	0.98	0.99	1.00	0.99	17	6	10.00	4.50	10.00	10.00
RL11	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	5	3	8.91	0.50	10.00	1.77
RL12	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	7	3	10.00	0.63	10.00	2.15
RL13	0.88	0.88	0.88	0.90	0.87	0.90	0.80	0.78	0.82	16	45	10.00	8.62	10.00	10.00
RL14	0.99	1.00	0.98	1.00	1.00	1.00	0.99	1.00	0.98	10	6	10.00	1.58	10.00	5.32
RL15	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	3	1.16	0.17	2.29	0.19
RL16	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8	5	10.00	1.38	10.00	3.20
RL17	0.99	0.81	0.88	1.00	1.00	1.00	0.99	0.81	0.88	17	11	10.00	5.09	10.00	4.56
RL18	0.99	0.45	0.63	0.99	1.00	0.95	0.99	0.50	0.67	13	6	10.00	9.33	10.00	4.56
RL19	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	11	7	10.00	5.87	10.00	5.62
RL20	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	7	4	10.00	1.24	10.00	2.07

with 0.68% and 0.94% accuracy, respectively, with the final counterexample or refinement occurring just before termination. In RL19, the low F1-score also indicated that the LSTM itself had not learned the language patterns.

For GRUs, 18 achieved 100% training accuracy, while two reached 99% accuracy and F1-score. Four models obtained 99% training but low F1-scores, again reflecting limited language generalisation, while three (T5, RL10, RL13) had both low accuracy and F1-scores. DFA extraction was fully successful for 23 of the 27 GRU models, each with 100% accuracy. The remaining four DFA extractions exceeded 90% accuracy and F1-scores, with reduced extraction accuracy correlating to weaker GRU training. Interestingly, in some cases, GRUs with less than perfect accuracy still produced 100% accurate DFAs.

The DFA extraction process was subject to a 10-second time limit. For LSTM models, most extractions continued until the limit, except for nine cases. Among these, 13 models completed their final refinement in under 1 second, and eight within 5 seconds, while only six required longer. A similar trend was observed for GRU models, where most extractions ran until the time limit, except for

Table 3. Performance of DFAs extracted from LSTMs using k-means clustering for initial partitioning. Columns represents: target language L, performance of the LSTM R compared to L, performance of the extracted DFA A_L compared to R, performance of A_L compared to L, number of initial partitions (clusters) k, total number of partitions in A_P, number of states $|Q_A|$ in A_L, total time t for the extraction (seconds), and the time $ref.t$ of the final refinement or counterexample (seconds). Each row represents languages for which the LSTM models were trained.

L	R *vs* L			A_L *vs* R			A_L *vs* L			k	$\|P\|$	$\|Q_A\|$	$t(s)$	$ref.t(s)$
	Acc.	*Prec.*	$F1$	*Acc.*	*Prec.*	$F1$	*Acc.*	*Prec.*	$F1$					
T1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	3	2	1.86	0.90
T2	0.99	1.00	0.93	1.00	1.00	1.00	0.99	1.00	0.93	3	7	4	10.00	1.08
T3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	5	5	4.08	2.26
T4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	4	4	2.88	1.25
T5	0.83	0.50	0.67	1.00	1.00	1.00	0.83	0.50	0.67	5	7	4	10.00	1.48
T6	0.99	1.00	0.99	1.00	1.00	1.00	0.99	1.00	0.99	4	9	4	10.00	2.16
T7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	9	5	10.00	2.11
RL1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	3	3	1.42	0.85
RL2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	3	3	1.28	1.03
RL3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	3	2	1.83	0.97
RL4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	3	2	1.83	0.99
RL5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	12	4	10.00	2.89
RL6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	11	5	10.00	6.04
RL7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	7	4	10.00	1.13
RL8	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	9	5	10.00	1.23
RL9	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	9	7	10.00	1.39
RL10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	7	3	10.00	1.49
RL11	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	3	3	1.40	0.91
RL12	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	6	3	10.00	1.45
RL13	0.99	0.99	0.99	0.68	0.75	0.64	0.68	0.75	0.64	4	11	19	10.00	9.73
RL14	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	5	4	3.50	1.18
RL15	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	6	3	10.00	2.96
RL16	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3	8	5	10.00	4.93
RL17	0.99	1.00	0.98	1.00	1.00	1.00	0.99	1.00	0.98	3	9	7	10.00	1.66
RL18	0.99	0.80	0.89	1.00	1.00	1.00	0.99	0.80	0.89	3	13	11	10.00	7.26
RL19	0.94	0.01	0.02	0.94	0.94	0.19	0.99	0.09	0.16	2	14	13	10.00	9.99
RL20	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2	8	4	10.00	4.26

six cases. Notably, nine GRU models reached their final refinement in under 1 second and ten within 5 seconds. Only RL13 and RL18 extended close to the full 10-second limit. In both cases, the remaining time after the final refinement or counterexample was spent verifying equivalence against all provided sequences. This suggests that termination is influenced by the number of strings evaluated for equivalence.

In experiments where k-means clustering was used to partition the state space, the extracted DFAs achieved accuracy comparable to those obtained from LSTMs without initial partitioning (3, 1). Since the sample size was small (27 models) and normality assumptions were not met, the non-parametric Wilcoxon signed-rank test was applied to compare extraction times between the two settings. Contrary to expectations, no significant improvement was observed ($p = 0.97$). Although k-means clustering reduced the number of refinements, the cost of computing initial clusters offset these gains, compromising overall efficiency.

The proposed algorithm is designed to terminate once equivalence is established between the L^* DFA A_L and RNN R. In practice, additional termination strategies were introduced during experiments (Section 4.5), based on the length

of the final counterexample and the number of excessive refinements. Crucially, these early termination conditions did not affect the correctness of the final DFAs, as the absence of counterexamples beyond these thresholds indicated a low likelihood of further discrepancies. Time constraints (10 seconds) also did not compromise correctness in most cases, with the exception of models RL13 and RL19.

Additionally, an experiment was conducted by substituting SVMs for hyperplane generation when partitioning the RNN state space, while keeping all other factors constant. In this setup, the SVMs separate the conflicted hidden states from others to form new partitions. The non-parametric Wilcoxon signed-rank test was then applied to both total extraction time and refinement/counterexample generation time for LSTM and GRU models.

For LSTMs, extraction with SVMs was significantly slower than with hyperplanes ($p = 0.004$ for both total extraction time and refinement/counterexample time). GRUs showed the same pattern, where extraction with SVMs was statistically slower than hyperplanes ($p = 0.046$ and $p = 0.042$, respectively), though the average time differences were smaller. Notably, total extraction time is not always a reliable measure, as many extractions ran until the time limit and termination was based on predefined factors.

When using hyperplanes, partitioning was refined upon detecting conflicts between hidden states, prioritising the separation of the most dissimilar states rather than isolating a single conflicted state. Subsequent refinements iteratively resolved remaining conflicts by grouping dissimilar states into distinct partitions, keeping the number of refinements minimal and ensuring computational efficiency.

7 Conclusion

This work presents an approach to extracting a DFA from an RNN to enhance the interpretability. The study employs the L^* algorithm for DFA extraction but introduces a novel approach to equivalence queries. The work in [32] proposed iteratively refining equivalence queries by comparing the L^*-learned DFA with an abstraction of the RNN state space generated by partitioning the RNN's state space using SVMs. Building on that, this work extends the methodology by using hyperplane-based state-space partitioning during equivalence checking. The main contribution of this work is the use of hyperplanes to partition the RNN's state space, ensuring the extracted DFA closely aligns with the RNN's decision-making process.

This method successfully extracts a sufficiently accurate DFA from a well-trained RNN. Similar to the work in [32], the algorithm needs only the language's alphabet and access to the RNN model; no additional details are required. The extraction process typically requires less than 10 seconds, as calculating hyperplanes to classify states in the abstract partitioning is computationally efficient.

If an RNN achieves a 100% training and testing accuracy, then the extracted DFA precisely represents the language on which the RNN was trained. Addition-

ally, for some RNNs that did not achieve perfect training and testing accuracy, the extracted DFA still captured the RNN's behaviour with 100% accuracy. However, in such cases, the extracted DFA deviates from the target language, indicating that the RNN itself did not properly learn the intended language.

Although for some RNNs the extraction process continued until the time limit of 10 seconds, the final refinement or counterexample generation was typically completed well before that. The remaining time was spent confirming equivalence using all provided strings up to termination. If additional termination strategies or constraints had been introduced, DFAs could have been extracted more efficiently. In this study, no limitations were imposed on the set of strings used for equivalence queries; instead, all strings were explored in lexicographic order until the time limit or another termination condition (Section 4.5) was reached. Thus, the termination was not triggered earlier until all strings were inspected within the given time limit.

The accuracy of the extracted DFAs refers to their similarity in predictions compared to the original RNN. If the DFA can produce outputs that are exactly or closely aligned with the RNN's predictions (where the RNN acts as the oracle during extraction), it can be concluded that the DFA has successfully approximated the RNN's behaviour. However, this does not imply that it has learned the exact internal behaviour of the RNN. Instead, the DFA captures the decision-making logic of the RNN—that is, how the RNN processes input sequences to generate outputs.

In this work, counterexamples are not prioritised, and no limit is imposed on the number of strings evaluated to check equivalence. Future work aims to explore methods for managing the termination of the algorithm more effectively, without exhaustively evaluating all available strings within the given time limit.

The results of this work align with the findings reported in [32]. More importantly, the extraction method is simpler, as it exclusively uses hyperplanes to partition the state space. The accuracy of the extraction remains comparable, but the key contribution of this study lies in employing a simpler refinement approach for partitioning. The L^* algorithm inherently scales, increasing computational time as the number of states grows. By reducing the time required for equivalence queries, the proposed method improves the efficiency of DFA extraction from RNNs.

Acknowledgments. Work supported by the EPSRC grant EP/T018313/1.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
2. Ayache, S., Eyraud, R., Goudian, N.: Explaining black boxes on sequential data using weighted automata. In: Proceedings of the 14th International Conference on Grammatical Inference. Proceedings of Machine Learning Research, vol. 93, pp. 81–103. PMLR (2018), http://proceedings.mlr.press/v93/ayache19a.html

3. Balle, B., Mohri, M.: Learning weighted automata. In: International Conference on Algebraic Informatics. Lecture Notes in Computer Science, vol. 9270, pp. 1–21. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-23021-4_1
4. Bojarski, M., Testa, D.D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., Zieba, K.: End to end learning for self-driving cars (2016), http://arxiv.org/abs/1604.07316
5. Chomsky, N.: Three models for the description of language. IEEE Transactions on Information Theory **2**(3), 113–124 (1956). https://doi.org/10.1109/TIT.1956.1056813
6. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. In: NIPS 2014 Workshop on Deep Learning (2014), http://arxiv.org/abs/1412.3555
7. Cleeremans, A., Servan-Schreiber, D., McClelland, J.L.: Finite state automata and simple recurrent networks. Neural Computation **1**(3), 372–381 (1989). https://doi.org/10.1162/neco.1989.1.3.372
8. Du, X., Xie, X., Li, Y., Ma, L., Liu, Y., Zhao, J.: Deepstellar: model-based quantitative analysis of stateful deep learning systems. In: Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 477–487. ACM (2019). https://doi.org/10.1145/3338906.3338954
9. Elman, J.L.: Finding structure in time. Cognitive Science **14**(2), 179–211 (1990). https://doi.org/10.1207/s15516709cog1402_1
10. Eyraud, R., Ayache, S.: Distillation of weighted automata from recurrent neural networks using a spectral approach. Machine Learning **113**(5), 3233–3266 (2024). https://doi.org/10.1007/s10994-021-05948-1
11. Giles, C.L., Miller, C.B., Chen, D., Chen, H.H., Sun, G.Z., Lee, Y.C.: Learning and extracting finite state automata with second-order recurrent neural networks. Neural Computation **4**(3), 393–405 (1992). https://doi.org/10.1162/neco.1992.4.3.393
12. Goldman, S., Kearns, M.: On the complexity of teaching. Journal of Computer and System Sciences **50**(1), 20–31 (1995). https://doi.org/10.1006/jcss.1995.1003
13. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735
14. Holzinger, A., Biemann, C., Pattichis, C.S., Kell, D.B.: What do we need to build explainable AI systems for the medical domain? (12 2017), http://arxiv.org/abs/1712.09923
15. Jacobsson, H.: Rule extraction from recurrent neural networks: A taxonomy and review. Neural Computation **17**(6), 1223–1263 (2005). https://doi.org/10.1162/0899766053630350
16. Kleene, S.C.: Representation of Events in Nerve Nets and Finite Automata, pp. 3–42. Princeton University Press (1956). https://doi.org/10.1515/9781400882618-002
17. Kodinariya, T.M., Makwana, P.R.: Review on determining number of cluster in k-means clustering. International Journal of Advance Research in Computer Science and Management Studies **1**(6), 90–95 (2013)
18. Marzouk, R., de la Higuera, C.: Distance and equivalence between finite state machines and recurrent neural networks: Computational results. arXiv preprint arXiv:2004.00478 (2020). https://doi.org/10.48550/arXiv.2004.00478
19. Mayr, F., Yovine, S.: Regular inference on artificial neural networks. In: Machine Learning and Knowledge Extraction. Lecture Notes in Computer Science, vol. 11015, pp. 350–369. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-99740-7_25

20. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics **5**(4), 115–133 (1943). https://doi.org/10.1007/BF02478259
21. Okudono, T., Waga, M., Sekiyama, T., Hasuo, I.: Weighted automata extraction from recurrent neural networks via regression on state spaces. Proceedings of the AAAI Conference on Artificial Intelligence **34**(04), 5306–5314 (2020). https://doi.org/10.1609/aaai.v34i04.5977
22. Oliva, C., Lago-Fernández, L.F.: On the interpretation of recurrent neural networks as finite state machines. In: Artificial Neural Networks and Machine Learning – ICANN 2019. Lecture Notes in Computer Science, vol. 11727, pp. 312–323. Springer Verlag (2019). https://doi.org/10.1007/978-3-030-30487-4_25
23. Omlin, C.W., Giles, C.: Extraction of rules from discrete-time recurrent neural networks. Neural Networks **9**(1), 41–52 (1996). https://doi.org/10.1016/0893-6080(95)00086-0
24. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., Devito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. Curran Associates Inc. (2019)
25. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
26. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press (10 2009). https://doi.org/10.1017/CBO9781139195218
27. Siegelmann, H.T., Sontag, E.D.: On the computational power of neural nets. In: Proceedings of the fifth annual workshop on Computational learning theory. pp. 440–449. ACM (1992). https://doi.org/10.1145/130385.130432
28. Tomita, M.: Learning of construction of finite automata from examples using hill-climbing RR: Regular set recognizer. Tech. Rep. CMU-CS-82-127, Carnegie-Mellon University (1982), http://shelf2.library.cmu.edu/Tech/8854663.pdf
29. Wang, Q., Zhang, K., II, A.G.O., Xing, X., Liu, X., Giles, C.L.: An empirical evaluation of rule extraction from recurrent neural networks. Neural Computation **30**(9), 2568–2591 (2018). https://doi.org/10.1162/neco_a_01111
30. Wei, Z., Zhang, X., Sun, M.: Extracting weighted finite automata from recurrent neural networks for natural languages. In: Formal Methods and Software Engineering. Lecture Notes in Computer Science, vol. 13478, pp. 370–385. Springer (2022). https://doi.org/10.1007/978-3-031-17244-1_22
31. Weiss, G., Goldberg, Y., Yahav, E.: Learning deterministic weighted automata with queries and counterexamples. In: Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019). https://doi.org/10.48550/arXiv.1910.13895
32. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples (extended version). Machine Learning **113**(5), 2877–2919 (2024). https://doi.org/10.1007/s10994-022-06163-2
33. Zeng, Z., Goodman, R.M., Smyth, P.: Learning finite state machines with self-clustering recurrent networks. Neural Computation **5**(6), 976–990 (1993). https://doi.org/10.1162/neco.1993.5.6.976