Neural Network Based Grammatical Learning and its Application for Structure Identification

Wang Xiangrui\textsuperscript{1}, Narendra S. Chaudhari\textsuperscript{2}

School of Computer Engineering, Block N4-02a-32, Nanyang Avenue, Nanyang Technological University, Singapore 639798
Emails: 1:xray@pmail.ntu.edu.sg; 2:asnarendra@ntu.edu.sg

Abstract. Structure identification has been used widely in many contexts. Grammatical Learning methods are used to find structure information through sequences. Due to negative results, alternative representations have to be used for Grammatical Learning. One such representation is recurrent neural networks. Recurrent neural networks are proposed as extended automata. In this chapter, we first summarize related works in grammatical inference and recurrent neural networks, and then propose a structural identification method to construct $k$\textsuperscript{th} order Markov Chains by using recurrent neural networks.

Keywords: Grammatical inference; Recurrent neural networks; Mixed $k$\textsuperscript{th} order Markov chains; Structure identification.

1. Introduction

As a special kind of neural network model, recurrent neural networks have feedback in their structure. By the feedback, recurrent neural networks are enabled to learn information that is normally learned by state machines. In this way, recurrent neural networks can do the work of automata and learn grammatical information. Omiln and Giles [56] construct Finite State Automata using recurrent neural networks. Sakakibara [59] construct generalized hidden Markov models with distributed representations using simple recurrent networks. Because there are strong negative results showing the impossibility of learning grammatical information from traditional learning models and representations, neural networks becomes a promising approach.

Structural identification is needed to be done in many fields, such as, speech processing, adaptive signal processing, time series prediction, biological sequence analysis, etc. The grammatical information in the sequence could be of value, since computational grammatical rules can perform a lot tasks and are more understandable than black box methods. By combining recurrent neural networks and grammatical inference techniques, we can avoid falling into the conditions in the negative results and still have the goodness of generating grammatical rules from the sequences, which could be used for identification or prediction.
We identify the problem of learning mixed $k^{th}$ order Markov chains. Since the existing methods of hidden Markov models or Markov chains only capture the information of fixed adjacent symbols in the sequences or need arbitrary model structure. In this chapter, we use recurrent neural networks to learn mixed $k^{th}$ order Markov chains. Our methods will capture all necessary chain information and do pruning during the learning process. The result will be mixed $k^{th}$ order Markov chains, whose structure is constructed automatically.

We first give a summary of grammatical inference in section 2. A survey of learning context-free language is given in section 3. An overview of various structures of recurrent neural networks is presented in section 4. In section 5, we give our method for learning mixed $k$th order Markov chains by recurrent neural networks. Finally, concluding remarks are given in section 6.

2. Grammatical Learning

Grammatical Learning, or Grammatical Inference, is to identify a computational grammar of a formal language from the given examples of the target language. After decades of research, Grammatical Inference has become a well-established research field in Artificial Intelligence. It is used in machine learning, computational learning theory, pattern recognition, computational linguistics, neural networks, formal language theory, information theory, etc. Over the past decade, researchers in these fields were brought together on conferences and workshops to share their ideas on grammar inference. The international conferences on Grammatical Inference (ICGI) took place in Britain (1993), Spain (1994), USA (1998), Portugal (2000), and Netherland (2002). Some results have been found important applications in natural language processing [1, 2], computational biology [3, 4], and related areas.

The research area of Grammatical Inference includes the following:

1. Different grammar induction models. These models include learning from examples, learning by queries, incremental learning and non-incremental learning, learning under distribution-free condition, learning under distributional assumptions. Other models are designed to generate learnability results, complexity results, measures of representation, and other properties of learning methods.

2. Algorithms for learning various classes of languages and automata. The languages include regular languages, context-free languages and context-sensitive languages, and their useful subsets under some syntactic constraints. Besides the traditional automata that accept languages in the Chomsky hierarchy, other models of grammars or languages are also studied. These variant models include tree and graph grammars, picture grammars, multi-dimensional grammars, attributed grammars, parameterized models, etc.

3. Methods used in grammar inference. Different approaches have been used for grammar inference. They include: artificial neural networks, statistical methods, symbolic methods, information-theoretic methods, minimum description length, complexity-theoretic approaches, heuristic method, etc.
4). Broader perspectives. This category includes the acquisition of grammar in conjunction with language semantics, semantic constraints on grammars, neural models of language, learning of languages that describe objects and events in space and time.

5). Applications. Potential applications of grammar inference are found in information retrieval, data mining, knowledge discovery, information extraction, natural language processing, bioinformatics, structural pattern recognition, database query processing and translation, text processing, and adaptive intelligent agents.

There are several surveys for Grammatical Inference. In Yasubumi Sakakibara’s article [5], fundamental methods are reviewed with an emphasis on Sakakibara’s work. In Lillian Lee’s article [6], the learning of Context-free language (CFG) is given a focused survey.

Much of the work in Grammatical Inference was related to the framework of Computational Learning Theory. An introduction to this framework is given in Laird’s paper [7]. A survey and related bibliography is given in Angluin’s paper [8].

Other surveys include: an introduction to Grammatical Inference [9], a survey on inductive inference [10], and Pitt’s paper [11] on inference of deterministic finite automata.

In the field of grammar inference, there are mainly three established formal learning models that learn from examples. One model is the identification in the limit by Gold [12], another model is the query learning by Angluin [13] and the PAC learning model by Valiant [14]. The three models have different criteria and descriptions of learning. We will introduce them briefly in this part.

2.1. Identification in the Limit

In the identification in the limit model, an infinite sequence of examples of the unknown language $L$ with grammar $G$ is given to the learning algorithm $M$ and the behavior of the algorithm after some finite time is used to describe the criterion of successful learning. A complete presentation of grammar $G$ is an infinite sequence of ordered pairs $(w, l) \in \Sigma^* \times \{0, 1\}$ ($\Sigma$ is the alphabet of the target language) such that $l=1$ if and only if (iff) string $w$ is in the language generated by grammar $G$, and every string $w$ of $\Sigma^*$ appears at least once in some pair in the sequence. At each time $t$, the learning algorithm $M$ receives an information unit (usually an example) $i_t$ of a complete presentation of $G$, and outputs a hypothesis $H(i_1, \ldots, i_t)$. A learning algorithm is considered successful if, after a finite amount of time, the hypothesis that the algorithm guesses is equivalence to $G$ at some point, and stops changing when more examples are taken as input. If for every complete presentation of the unknown grammar $G$, a learning algorithm $M$ successfully learns $G$, then $M$ is called to identify $G$ in the limit from complete presentations.

2.2. Query Learning

When human learns, the access to experimental results is sometimes useful. The same idea is applied to grammar inference. In the query learning model, the learning
algorithm has access to oracles given by a teacher that can answer certain types of questions about the target language. In some stages of the running process of the learning algorithm, query results can be used to aid the learning of the target grammar. There are mainly two types of queries that are widely used in this field:

1). **Membership Query.** The learning algorithm generates a string \( w \in \Sigma^* \). The teacher returns “yes” as the query result if \( w \) is a member of the language generated by the target grammar. Otherwise, the query result is “no”.

2). **Equivalence Query.** The learning algorithm generates a grammar \( G' \). \( G' \) is usually the grammar that the learning algorithm guesses. Then, the teacher returns “yes” as the query result if the language generated by \( G' \) is equal to that by \( G \), i.e. \( G' \) is equivalent to \( G \). Otherwise, the query result should be “no”, and it will also return a counter-example.

Let \( L(G) \) and \( L(G') \) denote the languages generated by \( G \) and \( G' \), respectively. A counter-example is a string \( w \in \Sigma^* \) such that, either \( w \in L(G) \) and \( w \notin L(G') \), or \( w \notin L(G) \) and \( w \in L(G') \). In other words, a counter-example is an evidence of difference between \( G \) and \( G' \). A formal definition of learning by equivalence queries is given in [15]:

Let \( R \) be a representation of a class of languages \( \mathcal{L} \), and let \( r(L) \) be any representation for a language \( L \in \mathcal{L} \). An equivalence oracle for \( L \in \mathcal{L} \) takes as input a \( r(L) \in R \); it returns “yes” if \( L \) is equivalent to \( L' \) (means that for any string \( w \in \Sigma^* \), \( w \in L \) iff \( w \in L' \), denoted \( L \equiv L' \)), and returns a string in the symmetric difference of \( L \) and \( L' \) otherwise. A deterministic algorithm \( M \) exactly identifies \( R \) using equivalence queries in polynomial time iff there is a polynomial \( p(., .) \) such that for all \( L' \), when \( M \) is run with access to an equivalence oracle for \( L' \), at any point in its computation, \( M \) has used an amount of time bounded by \( p(l, m) \), where \( m \) is the maximum length of any counter-example returned so far and \( l \) is the length of the shortest representation of \( L' \).

Learning in the limit model and query learning model are exact learning models, while the following one is a probabilistic learning model.

### 2.3. PAC Learning of Grammars

Valiant introduced the PAC learning model in [14]. In the context of grammar inference, the setup will be as follows. Random samples are generated independently from the domain \( \Sigma^* \) according to some probability distribution \( D \) which is arbitrary and unknown. The learning algorithm takes such a sample as input and then outputs a grammar. The success criterion in this model is in terms of the two parameters: accuracy parameter \( \varepsilon \) and the confidence parameter \( \delta \). These parameters are set for the learning algorithm. We call a learning algorithm PAC learns the target grammar \( G \), if the resulting grammar \( G' \) output by the learning algorithm can predict the membership of strings correctly with high probability (at least \( 1-\delta \)) and low error rate (less than \( \varepsilon \)).

### 2.4. Negative Results of Learning
Unfortunately, a large number of negative results for grammar inference are known, and we give some important results in this section.

**Negative Result for Identification in the Limit**

Gold has presented one main result that it is impossible to identify any of the four classes of languages in the Chomsky hierarchy in the limit if the data consists only of strings in the language being inferred [12]. A super-finite class of languages is one that contains all finite languages and at least one infinite language.

**Theorem 1.** There exists a super-finite class of languages that is not learnable in the limit from positive presentations.

Briefly, the proof is as follows. For a sequence of finite languages $L_1 \subset L_2 \subset \ldots$, let $L_\infty = \bigcup L_i$, $i=1,2,\ldots$. Suppose there is a learning algorithm $M$ that identifies $\{L\mid L \text{ is a finite language}\} \cup \{L_\infty\}$ in the limit. In other words, $M$ must identify $L_1$ in a finite amount of time. So, we specify the input as follows. First give $M$ enough examples (with repeats) from $L_1$ until it guesses $L_1$, or we can say that $M$ cannot identify $L_1$ that will lead to a contradiction. Then, give $M$ enough examples (with repeats) from $L_2$ until it outputs $L_2$, and so on. Clearly, all our examples will be in $L_\infty$. By continuing this way, given such a sequence of inputs, $M$ will make an infinite number of mistakes, and so $M$ cannot identify $L_\infty$.

**Negative Results for Query Learning**

Angluin designed a technique of approximate fingerprints for proving negative results for query learning models [15]. Using this technique, Angluin has shown that there is no algorithm using only equivalence queries identifies in polynomial time the class of deterministic finite automata (DFAs), nondeterministic finite automata, context-free grammars, or disjunctive or conjunctive normal form boolean formulas.

Negative results not only tell us what we cannot do, but also give us clues about what to do. Next, we will discuss some methods in learning of formal languages.

### 2.5. Inference of Regular Languages

In this section, we will use the classical definition of DFA, which is stated as follows.

**Definition 1.** A deterministic finite automaton is a quintuple $A=(\Sigma, Q, q_0, \delta, F)$, where $\Sigma$ is the alphabet and $Q$ are finite nonempty set of states, $q_0 \in Q$ is the starting state, $F \subseteq Q$ is the set of final state(s), and $\delta: Q \times \Sigma \rightarrow Q$ is the transition function. $\delta$ can be extended from $Q \times \Sigma$ to $Q \times \Sigma^*$ by $\delta(q, \lambda) = q$ and $\delta(q, aw) = \delta(\delta(q, a), w)$, $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$. The language recognized by the automaton $A$ is $L(A) = \{w \in \Sigma^* | \delta(q_0, w) \in F\}$. 
According to the negative results of learning from positive samples, learning DFA needs additional information. There are mainly three approaches in the grammar inference field on the learning of regular sets. One is learning regular sets from representative samples and using membership queries [16]. Another is identifying regular sets using equivalence queries and membership queries [17]. The third one is the inference of reversible languages [18].

2.6. Learning from Representative Samples

A live-complete set for regular language $L$ is any finite set of strings $P$ such that for every live (reachable and producible) state $q$ of $A=DFA(L)$ there exists a string $u \in P$ such that $\delta(q,u) = q$. A representative sample of $L$ is any finite subset $S$ of $L$ such that for every live (reachable and producible) transition $\delta(q,b)$ of $A$, there exists a string $u$ in $S$ that uses $\delta(q,b)$. An example of representative sample is shown in Figure 1.

![Figure 1. Representative sample. \{abbc, bad\} is a representative sample, but \{ac, bad\} is not a representative sample, since the transition $\delta(B, b)$ to state A is not used in the second sample.](image)

A new symbol $d_0$ is used to denote dead state. Let $P'=P \cup \{d_0\}$. Let $T'$ be the set of all elements of $P'$ and all elements of $f(u,b)$ (elements accessed by $(u,b)$) for all $(u,b) \in P \times \Sigma$. Let $T=T' \setminus \{d_0\}$.

The algorithm ID [16] first uses all the prefixes of representative sample strings as accessing strings, and creates each a state. Since this way may lead to redundant states in the acceptor, the algorithm uses membership queries to label distinguishing strings (or ending strings) for each state, and merges those states which can not be distinguished by any means.
The following is a formal description of the algorithm:

**Algorithm 1**  Algorithm ID:

//Initialization
- $E_0(d_0) = \text{null set}, v_0 (\text{the } 0\text{-th distinguishing string}) = \lambda$;
- For each $u \in T$,
  - if membership query about string $u$ returns “yes”,
    - let $E_0(u) = E_0(u) \cup \{\lambda\}$;
//Loop
- Once $E_i(u)$ has been constructed for all $u \in T^*$, search for a pair of elements $u, v \in P^*$, and a symbol $b \in U$, such that $E_i(u) = E_i(v)$ but $E_i(f(u, b)) \neq E_i(f(v, b))$.
- If find such a pair,
  - Choose some string $w \in E_i(f(u, b)) \oplus E_i(f(v, b))$, and define $v_{i+1}$ to be $bw$.
  - Set $E_{i+1}(d_0)$ to null set.
  - For each $u \in T$,
    - If query about $uv_{i+1}$ returns “yes”, $E_{i+1}(u) = E_i(u) \cup \{v_{i+1}\}$.

Next, we will illustrate the running of Algorithm ID with an example. Suppose the target automaton is the one shown below, and the given sample set is \{10111, 1101\}. We can see that this sample set is a representative sample of the target automaton.

![Algorithm ID: target automaton](image)

Algorithm ID first uses all the prefixes of the representative sample strings to construct an automaton as shown in Figure 3. Note that, each state has a corresponding accessing string, for example, the node marked ‘101’ has an accessing string of ‘101’ for the string ‘101’ can lead to this state. Similarly, string ‘110’ is the accessing string of the state marked with ‘110’. If there is a dead state, a self-loop is created for it.
The algorithm creates the extended states that could be reached by adding one symbol from the alphabet (Figure 4). The extended states are used to capture the transitions between the states of the prefixes.

Next, λ is used as the first distinguishing string. Here the word ‘distinguishing string’ means that by adding it to the accessing strings of states, some string can be accepted while others cannot. In this way, these strings make a different between the states. First, λ is used as suffix to connect with the accessing strings of all the states. And for each such a string, a membership query is proposed. For the state whose membership query is answered ‘accept’, λ is added to its distinguishing string set.
It can be seen from Figure 5 that, three states are marked, two of which are indicated by the representative sample. The other one, with accessing string ‘111’, is also marked because the target automaton accepts this string and hence the corresponding membership query is ‘yes’. No other state is marked in this stage, for their membership queries are answered ‘no’.

The next step is to find new distinguishing string. This step is an recursive operation. Based on the previous marking result, we can know that the states with different mark are definitely different because there exist a string that can distinguish them. Therefore, if two states, after a same transition, go to different labeled states, they also can be distinguished. We do not care much, if they are already carrying different mark. So the algorithm only searches the state pairs with the same label.

![Diagram](image1)

Fig.6. Algorithm ID: Step 4 - find new distinguishing string.

![Diagram](image2)

Fig.7. Algorithm ID: Step 5, final marking status
In Figure 6, state A and the dead state all labeled nothing. However, by the transition of 1, they lead to different labeled states: one is marked $\lambda$, the other is marked nothing. The algorithm will pick up these two states, and create the new distinguishing string this way: connect the transition and one distinguishing string from one of the marks that make the difference. In this example, 1, created as 1 connecting $\lambda$, is used as the new distinguishing string.

After this, the algorithm will use 1 as the distinguishing string for membership queries and marking. These are done repeatedly until no more state pairs can be found different, as shown in Figure 7.

Finally, the states with the same label will be ‘merged’, because they are not distinguishable at all and should be considered as the same state in the automaton. The ‘merge’ operation is to replace the merged states by a new single state; the new created state will substitute all the appearances of the merged states in the transitions. If this leads to multiple number of the same transition, delete redundant ones. This ‘merge’ operation can also be done in the ‘block’ sense as in some early works: consider the merged states as a whole block, and the previous transitions related to the merged states are considered as the block’s transitions.

After the ‘merge’ operation, the target automaton is learned as shown in Figure 8.

![Algorithm Id - learned automaton](image)

Fig.8. Algorithm Id - learned automaton

From the work of Algorithm ID, we have the following result:

**Theorem 2.** The class of deterministic finite automata can be identified in polynomial time from a representative sample and using membership queries [16].

### 2.7. Learning by Queries

It is also possible to learn DFA by membership queries and equivalence queries.

**Theorem 3.** The class of deterministic finite automata can be identified in polynomial time using equivalence queries and membership queries [17].

The details of the algorithm identifying regular sets using equivalence queries and membership queries can be found in Dana Angluin’s work [17]. A variant method can be found in [19]. The idea is to use the counter-example strings returned by
equivalence queries to maintain a tree of distinguishing strings, and thus find more states in the target automaton. The following is a formal definition.

**Definition 2.** Suppose $S$ is the access string set, an access string is denoted by $s$. $D$ is the distinguishing string set, a distinguishing string is denoted by $d$. $T$ is the classification tree of distinguishing strings and access strings. $M$ is the target automaton.

$M'$ is the learned automaton, and is updated during the algorithm. $\gamma$ is the counterexample string, which will get different accepting results in the target automaton and the guessed one. $\gamma[i]$ is an array of the 1st $i$ elements of $\gamma$. $\gamma_j$ is the $j$th ‘letter’ of $\gamma$. We use the algorithm described in [19] to illustrate the method. The following are procedures in the algorithm constructing DFA by membership queries and equivalence queries:

**Algorithm 2** Algorithm Learn-Automaton [19]:

### Procedure Sift($s, T$):
- Initialization: set the current node to be the root node of $T$
- Main Loop:
  --Let $d$ be the distinguishing string at the current node in the tree.
  --Make a membership query on $sd$. If $sd$ is accepted by $M$, update the current node to be the right child of the current node. Otherwise, update the current node to be the left child of the current node.
  --If the current node is a leaf node, then return the access string stored at this leaf. Otherwise, repeat the Main Loop.

### Procedure Tentative-Hypothesis ($T$):
- For each access string (leaf) of $T$, create a state in $M'$ that is labeled by that access string. Let the start state of $M'$ be the state $\lambda$.
- For each access state $s$ of $M'$ and each $b \in \{0,1\}$, compute the $b$-transition out of state $s$ in $M'$ as follows:
  --$s' \leftarrow\text{Sift}(sb, T)$
  --Direct the $b$-transition out of state $s$ to state $s'$
- Return $M'$
Procedure Update-Tree (γ, T):
• For each prefix γ[i] of γ:
  -- $s_i \leftarrow \text{Sift}(γ[i], T)$.
  -- Let $s'_i = M[γ[i]]$.
• Let $j$ be the least $i$ such that $s_i \neq s'_i$.
• Replace the node labeled with the access string $s_{j-1}$ in $T$ with an internal node with two leaf nodes. One leaf node is labeled with the access string $s_{j-1}$ and the other with the new access string $γ[j-1]$. The newly created internal node is labeled with the distinguishing string $γd$, where $d$ is the correct distinguishing string for $s_j$ and $s'_j$.

Main Procedure
• Initialization:
  -- Do a membership query on the string $λ$ to determine whether the start state of $M$ is accepting or rejecting.
  -- Construct a hypothesis automaton that consists simply of this single (accepting or rejecting) state with self-loops for both the 0 and 1 transition.
  -- Perform an equivalence query on this automaton, let the counterexample string be $γ$.
  -- Initialize the classification tree $T$ to have a root labeled with the distinguishing string $λ$ and two leaves labeled with access string $λ$ and $γ$.
• Main Loop:
  -- Let $T$ be the current classification tree.
  -- $M' \leftarrow \text{Tentative-Hypothesis}(T)$.
  -- Make an equivalence query on $M'$. If it is equivalent to the target then output $M'$ and halt. Otherwise, let $γ$ be the counterexample string.
  -- Update-Tree($γ$, $T$).
• Repeat Main Loop.

We now illustrate the running of the previous algorithm in a running example. Suppose the target automaton is the one shown in Figure 9.

![Algorithm Learn-Automaton: target automaton.](image-url)
The idea of the algorithm is to use the counterexample strings returned by equivalence queries to maintain a tree of strings, and thus find more states in the target automaton.

The running is shown in Figure 10. The algorithm will first ‘guess’ an automaton $M'$ (shown in Figure 10 (a)).

Fig.10. Algorithm Learn-Automaton: Running Example.
Then $M'$ is proposed in an equivalence query. Since the target automaton are different from the guessed one, a counter example ‘1103’ is returned, as ‘1103’ $\not\in L(M)$, but ‘1103’ $\in L(M')$. Then a tree (Figure 10 (b)) is constructed in this way: internal nodes are ending strings, leave nodes are accessing strings, because the accessing string $\lambda$ connecting ending string $\lambda$ is $\lambda$, it is rejected by the membership query, we put $\lambda$ at the left side, denote that it is rejected. For the returned counter example, since it connecting $\lambda$ is accepted by the membership query, we put it right, means that it is accepted.

After updating the tree, the automaton is constructed (Figure 10 (c)). The leave nodes are used as states and the transitions are constructed by membership queries. First, connect $\lambda$ and ‘0’, and do membership query on ‘0’, ‘0’ is not accepted. Therefore, ‘0’ goes left, the same leave node with $\lambda$, and so it points back to $\lambda$. Similarly, ‘1’ and ‘2’ also go left. When it comes to ‘3’, since it is accepted, ‘3’ goes right to the leaf nod of ‘1103’. (Figure 11) So, this transition points to ‘1103’. Other transitions are constructed in the same way.

![Fig.11. Algorithm Learn-Automaton: membership queries.](image)

Then, we get the new guessed automaton. This step is to construct tree from the automaton. After equivalence query, counterexample ‘012’ is returned, next is to find out the position that makes the difference between the target and the guessed one. By membership query, it finds out ‘2’ is the first letter makes the difference. Then the algorithm starts to update the tree $T$ (Figure 10(d)). It first does a membership query on the prefix ‘01’, ‘01’ is reject so the counter example goes left, and the suffix ‘2’ is put in the internal node, the prefix ‘01’ is put as its right child. The original string $\lambda$, is put on the left hand side. We do this recursively, until equivalence query be answered yes.

![Fig.12. ‘2’ is the first letter make the difference between $M$ and $M'$.](image)

### 2.8. Learning Subclasses of Regular Languages

In another approach of avoiding the negative results of learning regular languages, researchers have investigated the learning of subclasses of regular languages.
A $0$-reversible language is a language, which can be accepted by a DFA with only one start state and one final state, and when reversed, it is also a DFA. Likely, a $k$-reversible language is a language, which can be accepted by a deterministic automaton in terms of looking ahead $k$ letters.

Angluin has introduced the above $k$-reversible languages as a series of subclasses of regular languages [18]. In her paper, it is shown that a characteristic sample is a sufficient condition for identification from positive samples for $k$-reversible languages. A characteristic sample of an automaton $A$ of a $k$-reversible language is a finite sample $S \subseteq L(A)$ such that $L(A)$ is the “smallest” $k$-reversible language that contains $S$. It is shown that any characteristic sample is a representative sample.

Let $L$ be a finite language, define the prefix tree acceptor for $L$, $PT(L) = (\Sigma, Q, q_0, \delta, F)$, where $\Sigma$ is the alphabet, $Q = Pr(L)$, the set of all prefixes of $L$, $q_0 = \{\lambda\}$ if $L \neq \emptyset$, $q_0 = \emptyset$ if $L = \emptyset$, $\delta(u, a) = ua$, $u, a \in Q$.

For example, let $L = \{a, aa, aaa, baa\}$, $PT(L)$ is shown in Figure 13.

In short, algorithm $ZR$ first constructs the prefix tree of the given positive strings for a given language, and then it uses merger to satisfy the condition of zero-reversible language acceptor, and then get an automaton that accepts the target language.

Algorithm 3 Algorithm $ZR$:

```
//Initialization
• Construct the prefix tree PT(S) of the given sample $S$.
• Acceptor $A = PT(S)$
//loop
• From all states
  Find $B_1$ and $B_2$ such that $\delta(B_1, a) = \delta(B_2, a) = B_3$
  • If no such $B_1$ and $B_2$ exist, output and exit.
  Else merge $B_1$ and $B_2$
//end
```

Suppose we have the training sample $S = \{a, 0a, 01b, 011a, 1b, 10b\}$. We use this sample set on $ZR$. Firstly, $ZR$ constructs the prefix for the given sample $S$, as showed in Figure 14 (b). Then it merges the final states. (Figure 14 (c)) After steps of operations, the resultant automaton is constructed. (Figure 14 (d))
The algorithm ZR [18] is to learn a minimal $0$-reversible language acceptor from sample set $S$, while the algorithm learning a minimal $k$-reversible language acceptor is a generalized version of algorithm ZR.

**Theorem 4.** [18] The class of $k$-reversible language, for $k=0,1,2,…$, can be identified in the limit from positive presentation.

### 3. Inference of Context-free Languages

Since context-free languages are much more expressive, grammar inference of context-free languages are important. Therefore, many approaches on this topic have been made. Because of the negative results for learning languages, researchers have tried to avoid the impossibility by using variant conditions. Main approaches include learning by more helpful information, such as negative examples or structural information, and learning sub-classes of context-free languages.

![Fig.14. A running example of ZR algorithm.](image-url)
Definition 3. Let $G = (N, \Sigma, P, S)$, where $N$ is the set of nonterminals, $\Sigma$ is the alphabet, $P$ is the set of productions, and $S \in N$ is the start symbol. In this section, capital letters denote nonterminals, lowercase letters in the beginning of the alphabet denote terminals, lowercase letters at the end of the alphabet denote strings in $\Sigma$, and Greek letters represent strings in $(N \cup \Sigma)^*$. 

3.1. Learning from Structural Data

The idea behind this approach is to present structural information together with the string to the learning algorithm. The structural information is given in the form of a derivation tree of the strings.

Definition 4. $G$ is an operator grammar iff no production has adjacent nonterminals in its right-hand side, i.e., there are no productions of the form $A \rightarrow \alpha BC\beta$. The grammar $G$ is an operator precedence grammar iff $G$ is an operator grammar, and there are three binary relations $E$, $Y$, and $T$ on $\Sigma$ such that 1). The relation $E$ (equal in precedence) holds between all terminals that are adjacent or separated by a nonterminal in a derivation string. 2). The relation $Y$ (yield precedence) holds between the terminals preceding a handle and the leftmost terminal of the handle. 3). The relation $T$ (take precedence) holds between the rightmost terminal of a handle and the terminal immediately following the handle. 4). $E$, $Y$, and $T$ are pairwise disjoint.

A way to present structural information in a string form is called parenthesis grammar [20]. For a context-free grammar $G$, the corresponding parenthesis grammar $(G)$ is constructed by changing each production from $A \rightarrow \alpha$ to $A \rightarrow (\alpha)$ (suppose that parenthesis are not among the symbols in the $\Sigma$ of $G$). An incremental algorithm for identifying operator precedence grammars in the limit from positive parenthesized data is presented by Crespi-Reghizzi [21].

Structural data can be represented by skeletons, in which the nonterminal labels are removed from the derivation trees [22]. Under this form of representation, they are exactly the set of trees accepted by skeletal tree automata (SAs), which is a special kind of finite tree automata with removed non-leaf labels. An finite tree automaton $A$, when taking a tree $T$ as input, first assigns states to the leaves of $T$; when it runs, it moves up the tree by assigning each state to each node on the basis of the states of the node’s children according to the productions. If and only if it reaches a final state at the root of $T$, it is said that $A$ accepts $T$. Therefore, the problem of identifying CFLs from structured strings is reduced to the problem of identifying an SA.

By extending Angluin’s inference algorithm ZR [18] to SAs, Sakakibara presents a method for learning context-free languages from structural data in polynomial time [23]. A reversible context-free grammar is a CFG $G$ such that (1) $A \rightarrow \alpha$ and $B \rightarrow \alpha$ in the production set implies that $A = B$, and (2) $A \rightarrow \alpha B \beta$ and $A \rightarrow \alpha C \beta$ in the production set implies that $B = C$, where $A$, $B$, and $C$ are nonterminals, and $\alpha$, $\beta$ are the star set of the union of nonterminals and terminals.
Theorem 5. [23] The class of reversible context-free grammars can be identified in the limit from positive presentation of structured strings provided that the structured strings are generated with respect to a reversible context-free grammar for the unknown context-free language.

From the above theorem, we can see that the whole class of CFGs cannot be identified from positive presentation of structured strings.

Informally, the idea of the algorithm of learning CFGs from structured strings is: for each structured string, according to the parentheses, the algorithm extracts productions that form the derivation procedure, and assign states for them. If the algorithm finds somewhere in the production set exist (1) $A \neq B$, $A \rightarrow \alpha$ and $B \rightarrow \alpha$, or (2) $C \neq B$, $A \rightarrow \alpha B \beta$ and $A \rightarrow \alpha C \beta$, then, if situation (1) happens the algorithm will merge states $A$ and $B$, if situation (2) happens the algorithm will merge states $B$ and $C$.

Algorithm 4 Algorithm $RC$:

```plaintext
//Initialization
• Construct the grammar from the given structural data.
//loop
• From all states
    Find $B_1$ and $B_2$ such that $B_1 \neq B_2$ and
    $B_1 \rightarrow \alpha$ and $B_2 \rightarrow \alpha$
    or
    $A \rightarrow \alpha B_1 \beta$ and $A \rightarrow \alpha B_2 \beta$,
• If no such $B_1$ and $B_2$ exist, output and exit.
  Else merge $B_1$ and $B_2$
//end
```

We now illustrate the running of this algorithm

Suppose, the given structural data set is:

{ ( (girls) ((like) (dogs))), ( (girls) ((like) (cats))) }

Then, the grammar is constructed:

![Fig.15. Grammar from the structural data.](image)

Put in production form, suppose $S = P \_0$:

$$S \rightarrow P\_1, P\_2.$$
According to the rule of reversible context-free languages, \( P_1 \) and \( P_5 \) are to be merged, for we have \( P_1 \rightarrow \text{girls.} \) and \( P_5 \rightarrow \text{girls.} \). Similarly, \( P_3 \) and \( P_7 \) are to be merged. Suppose we use the smaller numbered state to represent the merged states. The following shows the result:

![Diagram: grammar after merging \( P_1 \) and \( P_5 \), \( P_3 \) and \( P_7 \)]

Next, because \( S \rightarrow P_1, P_2 \), and \( S \rightarrow P_1, P_6 \), \( P_2 \) and \( P_6 \) be merged.

![Diagram: Grammar after merging \( P_2 \) and \( P_6 \)]

Then, \( P_4 \) and \( P_8 \):

![Diagram: grammar after merging \( P_4 \) and \( P_8 \)]
After that, if put into production form, the production rule $P_4 \rightarrow \text{cats} \mid \text{dogs}$ is got. It shows that, in this way, some words in the same category is learned.

There are also other algorithms in this approach, such as, an algorithm being able to identify SAs in polynomial time by structural membership and structural equivalence queries [24], an algorithm similar to the one described above but needing “suitably selected” samples [25].

3.2. Learning Subclasses of CFLs

Commonly, researchers avoid the inlearnability described in Gold’s theorem by learning subclasses that do not contain all finite languages.

**Definition 5.** $G$ is $k$-bounded iff the right-hand side of every production contains at most $k$ nonterminals.

$k$-bounded CFGs are shown to be identifiable in polynomial time using equivalence queries and nonterminal membership queries [26]. Nonterminal membership queries give out a string $w$ and a nonterminal $A$; the answer is “yes” if $w$ is derivable from $A$ and “no” otherwise. Hence, from the nonterminal membership queries, the learning algorithm has access to the structure of the target grammar.

**Definition 6.** $G$ is a simple deterministic grammar iff all productions are of the form $A \rightarrow a\alpha$ where $\alpha \in \mathbb{N}^*$ and $|\alpha| \leq 2$, and if $A \rightarrow a\alpha$ and $A \rightarrow a\beta$ in $P$ then $\alpha = \beta$. Simple deterministic languages are languages accepted by a 1-state deterministic pushdown automaton with empty stack.

Ishizaka extends the above method to simple deterministic languages (SDLs) [27]. In Ishizaka’s method, nonterminal membership queries are no longer needed; instead, the algorithm uses extended equivalence queries, which propose a grammar $G$, while $G$ does not have to be a grammar for a SDL. The answer is “yes” if the target grammar is equivalent to $G$ and “no” otherwise. Extended equivalence queries are weaker than nonterminal membership queries, because they don’t convey structural information. However, since equivalence of CFGs is undecidable, the oracle answering the query must be quite powerful.

Yokomori gives a polynomial algorithm for learning SDLs that only proposes grammars of SDLs, but his algorithm requires prefix membership queries and deriv-
Prefix membership queries propose a string $w$, and the answer is “yes” if $w$ is a prefix of any string in the target language $L^*$. Derivative queries propose two pairs of strings $(u, v)$ and $(u', v')$, and the answer is “yes” iff $\{w \mid uwv \in L^*\} = \{w \mid u'wv' \in L^*\}$.

**Definition 7.** $G$ is linear iff all productions are of the form $A \rightarrow uBv$ or $A \rightarrow u$. $G$ is even linear if all productions are of the form $A \rightarrow uBv$, where $|u| = |v|$.

Takada has shown a way by which learning even linear languages (ELLs) can be reduced to the problem of learning regular languages [29]. He uses a control set to describe the possible derivations of strings in $L(G)$. Takada has proved that for any alphabet, there is a universal even linear grammar $G_0$ that generates any ELL over that alphabet with a control set that is regular. Hence, to learn an ELL, it should firstly learn the corresponding regular control set. The translation between control strings and strings in the language and between control sets and even linear grammars can be done in polynomial time, which means that equivalence and membership queries proposed by a regular language learning algorithm can be rewritten in terms of the corresponding even linear language. Extended result to linear languages can be found in [30], a similar result is given in [31].

**Definition 8.** $G$ is a pivot grammar (PG) iff $\Sigma$ can be partitioned into $\Sigma_o$ and $\Sigma_p$ such that all productions are of the form $A \rightarrow BaC, A \rightarrow Bb, A \rightarrow bB,$ and $A \rightarrow b$, where $a \in \Sigma_p$ and $b \in \Sigma_o$.

**Definition 9.** $G$ is very simple iff it is in Greibach normal form and for every terminal $a$, there is exactly one rule of the form $A \rightarrow aa$ in $P$. Very simple languages are a proper subset of SDLs. ($G$ is in Greibach normal form iff all productions in $G$ are of the form $A \rightarrow aa$.)

Algorithms that learn other subclasses of CFLs have been presented as follows, structurally reversible languages in [32], one-counter languages (languages accepted by deterministic one-counter automata) in [33], pivot languages in [34], and very simple languages in [35].

Bayesian approaches are also used to learn CFGs, which we will put in the next section “stochastic inference”.

### 3.3 Stochastic Inference

We now discuss the stochastic inference in this section. Stochastic modeling is rapidly growing for applications like natural language processing, speech recognition, and bioinformatics. A stochastic grammar has a probability for each production, and assigns a probability to each string that it generates. Hence, the grammar defines a distribution on the set of strings in the language. Hidden Markov models (HMMs) can be regarded as probability finite automata, which are widely studied and used
now. Stochastic context-free grammar (SCFG) is one level higher than HMM in the Chomsky hierarchy, and so that is a superclass of HMM.

**Definition 10.** $G$ is stochastic iff there is an assignment of weights, each weight being between zero and one inclusive, to the right-hand sides of productions such that for every nonterminal $A$, the weights of the right-hand sides of all the productions with $A$ on their left-hand sides sum to 1.

The learning of stochastic grammars from examples has two sub problems, one is to learn the structure of the grammar and the other is to estimate probabilistic parameters of the grammar. Estimating probabilistic parameters has efficient algorithms, which are based on expectation-maximization. There is forward-backward algorithm for HMMs [36] and inside-outside algorithm for SCFGs [37, 38]. Both methods increase the likelihood of the training sample and finally reach a local maximum. Therefore, the goodness of the result and the convergence property depend on initialization.

In the learning of structure aspect, Solomonoff has presented in his work [39, 40] an encoding method that convert grammars into strings, which will enable further assigning of *apriori* probabilities to strings and grammars in the same way. He also suggested a way to search the grammar space: based on current grammar at each step, finding nearby grammars recursively with higher goodness value.

Horning [41, 42] gives an algorithm that learns SCFGs in the limit with probability one from stochastic data (data derived by an SCFG). He also presents an idea of assigning *apriori* probabilities to grammars using a stochastic grammar for grammars. Other enumerative methods, which search all grammars with complexity less than a certain threshold, are presented by Wharton [43] and Van der Mude and Walker [44]. Unfortunately, all the above algorithms are inefficient. A hill-climbing algorithm [45] assigns a complexity measure for strings that has the following property: for strings of the same length and containing the same number of distinct symbols, if in a string all the symbols have the same frequency, the string is said to have the greatest complexity. Therefore, like the definition of entropy, the more random a string appears, the higher its complexity.

4. Neural Network Based Grammatical Inference

4.1. Introduction

Because the negative results of the basic models of grammar inference are based on the representation of the grammars, trying non-grammatical representations are natural to be investigated to find the solution. Since neural networks are intensively studied, the connectionist approach has been used for grammatical inference.
In the last decades, many approaches have been proposed to learning grammars by neural networks, because neural networks have been proved to have strong representational power and believed to be able to learn computational grammars. The idea is that, if we use both the input of the sequence and the status of the network, we can get a finite machine like neural network: the input acts as transition symbols in a finite automaton, and the statuses of the nodes in a network act as state in a finite automaton. In this way, the status of the neural network must be feedback, so this kind network is called recurrent neural networks.

4.2. Recurrent Neural Network Structures for Grammatical Inference

Since the idea of using recurrent neural networks for grammatical inference is to use the input and the previous running states to simulate the function of a finite state machine, the structures of recurrent neural networks are all fit in the following framework, which we call recurrent network. (Figure 19).

In this framework, an input is given to a machine, whatever it is. The input is usually a sequence, and is fed to the machine once a part at a time. While taking the input each time, the machine reacts to the input, sometimes generates output, and will change itself according to the input. Next time, when the input is given as a following part of the previous one, the machine will react according to this input and the feedback from the last time. In this way, the recurrent network acts like an automaton: it takes sequence input, memorizes some information, and subsequently, gets the result.

Any finite automaton can be fit into this framework of recurrent network. The recurrent neural network, while using neural network inside the machine, also fit into this framework, and thus is used to learn grammars.

In the terms of connecting methods, some structures are listed below.
• Frasconi-Gori-Soda (FGS) locally recurrent networks [46]. In this network structure, the hidden nodes will feedback only locally in the next time phase. It is also believed that FGS structure is incapable of doing the grammar-learning task. An example is shown in Figure 20.

![FGS Recurrent Neural Network Diagram](image)

**Fig.20.** A FGS recurrent neural network

• Narendra and Parthasarathy recurrent networks [47]. In this network structure, the output is given as feedback to the hidden layer after a time

![Narendra and Parthasarathy Recurrent Neural Network Diagram](image)

**Fig.21.** A Narendra and Parthasarathy recurrent neural network

• Narendra and Parthasarathy recurrent networks [47]. In this network structure, the output is given as feedback to the hidden layer after a time
delay. The difference between this structure and FGS is that, in this structure, all hidden nodes get the feedback from all the outputs, while in FGS, each hidden node only entertains one local feedback from the last step. An example is shown in Figure 21.

- **Elman recurrent networks** [48]. This network structure, the one in FGS is extended in a way that all hidden nodes get the feedback from the status of the last step. An example is shown in Figure 22.

![Fig.22. An Elman recurrent neural network](image)

- **Williams and Zipser fully recurrent networks** [49]. This network structure is called ‘fully’ because all the outputs and all the hidden nodes get the
feedback from all the outputs and all the hidden nodes. An example is shown in Figure 23.

For different input recursion, there are:

- **First-Order Recurrent Networks.** Frasconi et al. proposed this recurrent network structure for injecting DFAs [50,51]. As shown in the next Figure, the recurrence involved in this network structure only affects the hidden nodes. Therefore, each time, the network uses the information of current inputs and results from the last step.

![A first-order recurrent neural network.](image)

- **Second-Order Recurrent Networks.** Giles and Omlin proposed this recurrent network structure to represent partial prior knowledge and therefore improve convergence performance [52-55]. As shown in Figure 25, in this structure the feedback is given to the inputs, or external nodes (in contrast to the hidden nodes). In this way, recurrent neural networks using this structure are able to use combined information of adjacent input sequence positions.

![A second-order recurrent neural network.](image)
4.3. Constructing Recurrent Neural Networks to Learn Grammars

In this section, we will mainly introduce two approaches to learn grammars by recurrent neural networks. We call one explicit and the other implicit.

Explicit Methods.

These methods are same on the point that they all use some kind of straight encoding method to realize the conversion between grammatical rules and recurrent neural networks. Usually, one such method will take the input samples to construct the nodes and related parameters according to a converting rule. Thus, the nodes in the recurrent network represent explicit grammatical rules.

Under such a framework, recurrent neural networks constructed by these methods are easily converted to grammatical rules; yet, they may lose some strength of the neural networks.

Here, we use an way encoding grammars by neural networks given in [56] to illustrate the idea of these methods.

The structure used is a second-order recurrent neural network structure. Suppose the grammatical rule of a DFA we want to encode is as the one shown in Figure 26(a). In Figure 26(b), the detail of the encoding is shown: $I^t_k$ represents the input symbol $a_k$ at time $t$, $S^t_j$ represents the state $q_j$ with the transition $a_k$ as an out-arc at time $t$, $S^{t+1}_i$ represents the state $q_i$ with the transition $a_k$ as an in-arc at time $t$, while $I^{t+1}_k$, $S^{t+1}_j$ and $S^{t+1}_i$ represent basically the same symbol or states, but at time $t+1$. The training will include determining the weights $W_{ij}$, $W_{jk}$ and the biases $b_i$ and $b_j$. The value of the weight $W_{ij}$ connected to the node $S_i$ represents the acceptance of state $q_i$ (activated means accepted, inactivated means rejected). $H$ denotes the rule strength. $\otimes$ denotes the operation $S^t \cdot f^t$. 

Fig.25. A second-order recurrent neural network.
It is argued [48] that though explicit methods have many attractions, their drawbacks include: first, explicit presentation of grammatical rules while with limited resources, will vary depending on the specific case, and is hard to determine before constructing the following problems: what type of nodes or how many of them should be used; second, the learning results of explicit methods are likely to underestimate the richness of linguistic structures.

Therefore, instead of using explicit encoding, these methods use a fairly general expression [57]:

$$S_{i+1}^{t+1} = F(S_i^t, I^t, W)$$

where $F$ is a nonlinearity mapping the state node $S^t$ and the input $I^t$ to the next state $S^{t+1}$ at next time $t+1$. The weight matrix $W$ contains parameters of the mapping and is usually to be learned.

A first-order recurrent neural network mapping function is defined in [58] as:

$$= \sigma \left( \sum_j N_j S_j^t + \sum_k Y_k I_k^t \right)$$
where $N$ is the number of hidden state neurons and $L$ the number of input neurons; $W_{ij}$ and $Y_{ij}$ are the real-valued weights for respectively the state and input neurons; and $\sigma$ is a standard sigmoid discriminant function.

A second-order recurrent neural network mapping function is:

$$S^{t+1}_i = \sigma \left( \sum_{j,k}^{N,L} W_{ijk} S^t_j S^t_k \right) \Rightarrow \sigma \left( \sum_{j,k}^{N,L} W_{ijk} S^t_j I^t_k \right)$$

In the implicit methods, recurrent neural networks are constructed more like a neural network. It can be seen that when using these methods, the extracting of rules will be less obvious than using explicit methods.

4.4. Extracting Grammatical Rules from Neural Networks

In this section, we will introduce a method of extracting rules from recurrent neural networks using implicit methods, which is described in [57].

The rule extraction process consists of the following steps:
1) clustering of DFA states,
2) constructing a transition diagram by connecting these states together with the alphabet labelled transitions,
3) putting these transitions together to make the full digraph forming cycles, and
4) reducing the digraph to a minimal representation.

A simple way of finding these clusters [57] is to divide each neuron’s range [0,1] into $q$ partitions of equal size. For $N$ state neurons, $q^N$ partitions. For example, for $q=2$, the values of $S^t \geq 0.5$ are encoded as 1 and $S^t < 0.5$ are encoded as 0, and there are $2^N$ regions with $2^N$ possible values. Then, if before an input symbol is fed into the network, there is a set of activated hidden neuron partitions, then this set will be considered as the state carrying the out-arc of the input symbol. The set of activated hidden neuron partitions after the input symbol runs on the network is considered as the state reached by the transition carrying the input symbol from the previous state.

5. RNN for learning mixed $k$th-order Markov Chains

5.1. Introduction
In traditional framework of Markov Chains, the models are able to capture highly observed sequence patterns to help classify sequences. In the first-order Markov Chains, only two-length sequence patterns are captured, while in the second-order Markov Chains, three-length sequence patterns are captured. In the same manner, $k^{th}$ -order Markov Chains can and only can capture $k$-length sequence patterns.

In practice, there may not be an exact number of $k$ to determine. Highly observed patterns may include sequences of different length at the same time. Therefore, using all highly observed patterns in a minimal complexity manner is desired.

There are some works related Markov models with recurrent neural networks [59], however, the structure of the constructed model are unclear.

In this section, we will present our work of a recurrent neural network used to learn mixed $k^{th}$ -order Markov Chains.

5.2. The model

Consider the recurrent network in Figure 27.

![Recurrent Neural Network Structure](image)

**Figure 27.** Recurrent neural network structure for mixed $k^{th}$ order Markov models

The input $I$ is one symbol from the alphabet of a running sequence. At each time $t$, one symbol is put as the input to the recurrent neural network, and at the next time $t+1$, the following symbol will be put as input. In this way, the sequence is given to the neural network.

After taking the input symbol, a corresponding nodes representing the same symbol in $h_{0.1}^{(t)}$ will be activated. Here, $0$ means the $0^{th}$ order information of the Markov chain. We attach each node a purity score value calculated by the following function.

$$S_{ij} = \lg \left( \frac{Pr(h_{ij} | P)}{Pr(h_{ij} | N)} \right),$$

(4)
where, $P$ is the set of positive samples, and $N$ is the set of negative samples, $Pr(h_{ij} \mid P)$ denotes the probability of activating node $h_{ij}$ in the running of a positive sample sequence, while $Pr(h_{ij} \mid N)$ denotes the probability of activating node $h_{ij}$ in the running of a negative sample sequence. In practice, we use the weighted frequencies to represent the probabilities.

$$Pr(h_{ij} \mid P) = \frac{n_{ij}}{n_P},$$

$$Pr(h_{ij} \mid N) = \frac{n_{ij}}{n_N},$$

where, $n_{ij}$ is the number of activation of the node $h_{ij}$, $n_P$ and $n_N$ denote the number of positive and negative sequences respectively. Note that this does not promise normalization, but enough for the scoring. The $\log$ makes the score addable, and also simplify the result from some normalization factor, any base is acceptable, and usually we use 2.

If a node’s score is larger than a certain threshold $tr$, we consider the node be a ‘pure’ one, that is, it is enough for classification, and will need no further check. In figure 3.1, the shaded nodes are ‘pure’ nodes.

The recurrent neural network does classification this way:

- put the unclassified sequence from the first symbol
- activate the connected nodes in the hidden layer
- if the activated node is a ‘pure’ node
  - calculate the score $S$ for sequence as
    - $S = S + S_{ij}$
  - Reset all activated nodes to inactivated.
- Else
  - Copy the activating result of the hidden layer to the context layer
- Get the next symbol in the sequence, until reach the end.
- the sequence is positive, if $S$ is positive, and negative oth-

We also propose a way to construct the recurrent network, which is shown in the next section.

5.3. Construction of the model

Basically, our idea of using recurrent neural networks to learn mixed $k^{th}$ order Markov Chains is to first find small length highly observed patterns. When a short pattern is considered to be ‘pure’ enough, the longer sequences containing the short
pattern as a sub-pattern is no longer considered, because the short patterns are already enough for classification.

The construction of the recurrent neural network contains the following steps:

1). For each symbol in the alphabet $\Sigma$, create a node representing it. Suppose $\Sigma = \{a, b, c\}$ then, the sequences are input to the network. Originally, the hidden nodes are all non-pure nodes. As the sequence are input into the network symbol by symbol, the frequencies are calculated by adding one to the visit number each time the node is activated. Note that there are two such visit number, one for positive and the other for negative. After this, we calculated the scores of the nodes using (1). Then, mark those nodes whose score greater than the threshold $tr$ or less than $-tr$. Suppose this time the node representing symbol $a$ is marked as pure node.

![Fig.28. Recurrent neural network structure after the first scan.](image)

2). From the previous scan, we know which nodes are ‘pure’, that is, is considered to give enough confidence to make a prediction. Then, the non-pure nodes will be copied for two length patterns. In the example, $b'$ and $c'$ are copied as $b$ and $c$, shown in Figure 29.

![Fig.29. Recurrent neural network structure constructing 1st order context nodes.](image)

3). Further, four nodes are constructed to calculate first order information. Again, the sequences are input to the network. This time, the network uses the inputs in this way: if a 'pure' node is activated, the network reset all activated nodes to inactivated ones, in which way the pruning is done, since the shorter patterns...
will have higher priority than longer patterns. If not, the context nodes are copied from the corresponding hidden nodes. New input is input. Still, if a ‘pure’ node is activated, the network is reset. If not, the hidden nodes combined the context nodes will activate new constructed nodes. Again, the frequencies are calculated and after the scan, ‘pure’ nodes are marked. After the 2\textsuperscript{nd} scan, the recurrent neural network is like the one shown in Figure 30.

![Figure 30. R neural network after the second scan.](image)

4). After this, the network is going to run in the same manner, until in one step, all the new constructed nodes are ‘pure’, or after the \((k+1)\textsuperscript{th}\) scan. The limit number \(k\) are use to avoid too many nodes are constructed.

In the third scan, the network of the example will copy \(bc^t\) to context nodes. In the next scan, get the following network.

![Figure 31. Final recurrent neural network after scans.](image)

5.4. Discussions

It can be seen that, we have presented an explicit method to construct recurrent neural networks for mixed \(k\textsuperscript{th}\) order Markov Chains, which captures all useful (in terms of classification) highly observed under \(k\)-length patterns.
Though in the worst case, the network will have $O(|\Sigma|^{k+1})$ nodes, and $O(|\Sigma|^{k+1})$, in practice, by limiting $k$, the nodes will be in a reasonable number. We can also see that the time complexity, if by serial algorithm simulation, is $O(k n |\Sigma|^{k+1})$ where $n$ is the input length. Here $k$ and $|\Sigma|^{k+1}$ are constant if the model has been determined, so the time complexity is only linear with the input length $n$.

6. Concluding Remarks

Structure Identification is a main problem in many areas. Grammatical inference methods are used to find structural information among sequences and use to do further identification. Since negative results exist in the field, grammatical inference models must be aided with alternative representations or information. In this chapter, we first gave an overview of existing grammatical inference methods, which are mainly based on query learning, subclass learning, and learning by other helpful information. As recurrent neural networks can perform tasks that are normally done by automata, they are good alternative representations to use against the negative results in traditional models. Following the introduction of grammatical inference, we give a brief summary of recurrent neural networks.

We also presented a method to learn $k^{th}$ order Markov Chains by training recurrent neural networks through the given sample sets using a constructive method with time complexity $O(k n |\Sigma|^{k+1})$. By using recurrent neural networks model, we avoid the negative results caused by traditional representations of automata. In this way, the process is able to generate the learning results and can easily extract $k^{th}$ order Markov Chains from the result recurrent neural networks.

References


[16] Dana Angluin, A Note on the Number of Queries Needed to Identify Regular Languages, Information and Control 51, 1981, pp. 76-87.


