A Neural Model for Regular Grammar Induction

Peter Belcak  
*ETH Zürich*  
Zürich, Switzerland  
belcak@ethz.ch

David Hofer  
*ETH Zürich*  
Zürich, Switzerland  
davhofer@ethz.ch

Roger Wattenhofer  
*ETH Zürich*  
Zürich, Switzerland  
wattenhofer@ethz.ch

Abstract—Grammatical inference is a classical problem in computational learning theory and a topic of wider influence in natural language processing. We treat grammars as a model of computation and propose a novel neural approach to induction of regular grammars from positive and negative examples. Our model is fully explainable, its intermediate results are directly interpretable as partial parses, and it can be used to learn arbitrary regular grammars when provided with sufficient data. Our method consistently attains high recall and precision scores across a range of tests of varying complexity. We make the detailed results and code readily available.1

Index Terms—neural networks, regular languages, grammar induction, program synthesis

I. INTRODUCTION

Finite automata, or finite state machines, are the simplest class of computational devices possessing memory. Their memory is finite and stored exclusively as their state, but that alone is enough to make their analysis challenging. In contrast with sufficient accuracy. For instance, L

induction, program synthesis

The learning of finite automata is a classical problem in computational learning theory, and can be phrased as follows: given a language L and a set of examples E, use E to find a finite acceptor automaton D that accepts words in L and rejects words not in L, or at least find a D that performs these duties with sufficient accuracy. For instance, L can be the language generated by the regular expression a*bb*. The set of examples E could consist of the entire L, with us placing an additional requirement of conciseness on the learning process to arrive at a single, practical solution (say an automaton with at most three states). Or, E could contain words such as aab and bbb marked as positive examples (the words D should accept), and aoo and aba marked as negative examples (words to be rejected by D).

Strictly algorithmic, formal, statistical, and genetic approaches have all been proposed, but this problem remains open mainly because the solutions lack an agreed single measure of merit. Some have argued for automata that concisely represent E, while others preferred automata that generalised well to L in scenarios where E contains only a few examples representative of the original language. Another degree of freedom arises in the constraints that are placed on E. One can insist that E consists of only positive examples (i.e. E ⊆ L), that E contains enough examples to sanction good generalisation to L (e.g. all repetitions of a pattern beyond a certain count signify that the pattern may be repeated indefinitely), or that E contains both positive and negative examples (i.e. E ∩ L ∩ L′, where L′ denotes words not in L). We give a detailed overview of related work in Sect. II.

This paper proposes a neural model that learns a finite acceptor automaton for target language L from a given E consisting of positive and negative examples. We insist that the logic of the resulting automaton is fully explainable – fully comprehensible by humans, that the resulting automata generalise well to L despite being based on only E, and that the representations are concise, or at least such that they do not contain too much redundancy of computational logic.

Internally, our model learns a regular grammar rather than a finite automaton. Briefly, a grammar G is a triplet of AT, AN, P – the terminal alphabet, non-terminal alphabet, and set of productions, respectively, where one of the letters of AN is marked as the start symbol (the “root” of G). A left-regular grammar is a grammar in which all productions of P are of the form A → c, A → c, or A → Bc for A, B ∈ AN, c ∈ AT, the empty string. A right-regular grammar uses productions of the form A → cB instead of A → Bc. Regular grammars are equivalent to finite automata in their language-generating power and admit straightforward conversions into each other. We do not consider empty languages or grammars that give empty languages.

In our setting, there is a target language L = ⟨G⟩ generated by the ground-truth grammar G. We are shown examples E taken from both L and L′, and our model internally learns a hypothesis grammar G′. In other words, we do not train neural automata to become accurate acceptors for given languages L, but work instead with a fully explainable regular language learning parser under an acceptor training setup. We use less information than many previous methods for grammatical inference and let the grammar emerge under single-bit supervision.

Our contributions are:

- We introduce of a novel neural model tailored specifically to the learning of regular grammars from positive and negative acceptor examples. There are no limits on the complexity of the regular grammars it can learn.

---

1We make our code available at https://github.com/pbelcak/neregrain
• We describe in detail a procedure for the recovery of the learned grammars and parse trees from the internals of our model, warranting explainability.
• We systematically evaluate our model across the dimensions of grammar size, grammar complexity, and training data quantity.

II. RELATED WORK

From the perspective of algorithmic learning, we divide the literature on learning of regular languages into two groups: one concerning the learning of finite automata and the associated regular expressions, the other addressing the problem of grammar induction.

Learning finite automata. Early work on the subject leveraged Hidden Markov Models [15] and probabilistic finite state machines [5] as the models for learning. Later approaches readily recognized the utility of regular expressions as a form for description and tended to be deterministic. Polynomial-time algorithms were proposed for learning of regular expressions without union operation from chosen classes of positive examples [3] and for learning of unambiguous regular expressions of maximum loop depth 2 [13]. A genetic programming approach leveraging both positive and negative examples was used in [18], and a further polynomial-time method for 1-unambiguous regular expressions aiming for simplicity of the resulting expressions was presented in [9]. Most of the recent work on regular expression learning takes the line of natural language processing (NLP) and proposes methods for particular uses in real-world datasets. A genetic programming method for text extraction from XML documents is outlined in [2]. [14] focuses on identifying email campaigns with high precision while phrasing it as an optimisation problem, and [4] gives a method based on Support-Vector Machines tailored to clinical texts.

Note that all of the above work either fences out a particular sub-class of regular expressions that are to be learned (e.g. union-less or 1-unambiguous) or makes additional assumptions based on the nature of the particular real-world problem it is designed to solve.

Grammar induction. Pioneering work on the learning of grammars from examples often attempted to construct probabilistic context-free grammars that generated the target language [9]. Subsequent attempts employed a wider variety of tactics including Bayesian methods [6] and genetic programming [19]. The learning was done chiefly from positive examples, though methods for automatic negative example generation from positive examples were later introduced [17]. Similarly to above, contemporary work on grammar induction is almost exclusively guided by the desired applications in NLP rather than interest in theory of computation. As such, black-box recurrent neural automata [7], reinforcement learning models [20], and partially interpretable structured attention [12] and transformer [10] models dominate the sub-field at present.

Our approach spans both of these groups. We train on accept/reject information just like an acceptor automaton but learn a grammar. In contrast to research on learning finite automata and regular expressions, there are no inherent limits on the complexity of grammars our model can learn – it can learn any regular grammar, not a sub-class characterised by a particular type of regular expressions. Further, in contrast with the recent efforts in grammar induction, the neural architecture we present learns a grammar in a fully explainable fashion. Explainability is present in the training procedure, grammar extraction, and the use of our model for direct parsing by the learned grammar.

Given the above, our approach is more akin to research in program synthesis and fits better under the paradigm of “algorithm learning”. This is because we can view the language examples together with the annotations that mark them as positive and negative as input-output pairs for the broader programming by example (PbE) task [16]. In PbE, a program in a given programming language is to be synthesised based on a set of input-output pairs. In our case, the programming language is the rewriting scheme for regular grammars and the program is the particular regular grammar that generates \( \mathcal{L} \) or a substantial overlap therewith.

III. MODEL

Our network consists of three components: Terminal Grammar Unit, Non-terminal Grammar Unit, and Start Selector Unit. These are engaged sequentially following a simplified CYK algorithm to perform the function of an acceptor for the target language \( \mathcal{L} \). After training, the internals of each unit can be directly inspected to recover the learned regular grammar \( \mathcal{G} = (\mathcal{A}_T, \mathcal{A}_N, \mathcal{P}) \). Note that while the terminal alphabet \( \mathcal{A}_T \) is considered known (from \( \mathcal{L}, \mathcal{E} \)) and shared with our model, the non-terminal alphabet \( \mathcal{A}_N \) is learned indirectly through use in learned productions \( \mathcal{P}' \). We present our method as for the learning of left-regular grammars, but the entire setup can be inverted to produce right-regular grammars.

A. Application Algorithm

For each example word \( w : a_1 a_2 a_3 \ldots a_{\ell} \) we one-hot encode letters \( a_i \) into \( \ell \)-dimensional vectors \( v_i \), where \( \ell = |\mathcal{A}_T| \) is the size of the terminal alphabet.

If \( w \) is a positive example we associate it with label \( y = 1 \), otherwise \( y = 0 \). \( v_1 \) is then fed into the Terminal Grammar Unit to produce an intermediate parse \( \mathcal{P}' \)-dimensional (belief) vector \( u_1 \). \( n' \) is a parameter of the training, giving an upper bound on the number of non-terminals that may appear in \( \mathcal{P}' \). The ground-truth non-terminal alphabet \( \mathcal{A}_N \) is not known to us during training and so neither is its size \( n = |\mathcal{A}_N| \). \( n' \) is therefore just a guess for \( n \), made in a thought process similar to that for the number of latent dimensions to be used in a disentangling variational autoencoder.

For \( 1 \leq i < \ell \), \( v_{i+1} \) and \( u_i \) are then provided as the input for the Non-terminal Grammar Unit, which in turn produces the intermediate parse vector \( u_{i+1} \).

Finally, \( u_{\ell} \) is processed by the Start Selector Unit, which outputs a number \( o(w) \) between 0 and 1 representing the belief that \( w \in \mathcal{L} \). The whole procedure is illustrated in Fig. 1 on
the unit level, and in Fig. 2 with internals and on a simple example. In training, \( o(w) \) is assigned a loss value computed as the binary cross-entropy between \( y \) (true label) and \( o(w) \) (predicted label). The model is trained to minimise total loss combining the cross-entropy and two other losses reflecting the quality of the hypothesis grammar.

B. Terminal Grammar Unit (TGU)

The TGU serves as the hypothesis grammar parser for potential terminal productions. It takes a \( t \)-dimensional vector \( v_1 \) — the one-hot encoding of the first letter — as input, uses it to query the trained hypothesis terminal production \( n' \)-by-\( t \) matrix \( P_T \), and clamps the output between 0 and 1:

\[
\text{TGU}(v_1) := \text{clamp}(\sigma(P_T) v_1),
\]

where \( \text{clamp}(v) := \max(\min(1, v), 0) \) and \( \sigma \) is the logistic sigmoid, each applied element-wise. If \( A \) and \( B \) are one-hot-encoded as the \( i \)-th and \( j \)-th euclidean basis vectors, then the \( i, j \)-th entry of \( \sigma(P_T) \) represents the belief of the model (the strength of TGU’s hypothesis) that the production \( A \rightarrow B \) is a part of the grammar \( G' \) being learned from \( E \).

C. Non-terminal Grammar Unit (NGU)

The NGU is the recurrent unit of our architecture and is the parser for the hypothesis grammar’s non-terminal productions. It holds \( n' \) \( n' \)-by-\( t \) trained matrices, each representing the hypothesised non-terminal productions with one of the \( n' \) potential non-terminals \( A_N' \) on the left-hand side. To perform its parse, the NGU takes the previous parse vector \( u_t \), current terminal vector \( v_{i+t} \), and yields \( u_{i+t} \) where the \( k \)-th entry of \( u_{i+t} \) for \( 1 \leq k \leq n' \) is given by:

\[
\text{NGU}(u_t, v_{i+t}) := \text{clamp}((\sigma(P_N^k) v_{i+t})^T u_t).
\]

If \( A, B \), and \( c \) are one-hot-encoded as the \( k \)-th, \( i \)-th, and \( j \)-th euclidean basis vectors respectively, then the \( i, j \)-th entry of \( \sigma(P_N^k) \) represents the strength of NGU’s hypothesis that the production \( A \rightarrow Bc \) belongs to \( P' \).

D. Start Selector Unit (SSU)

The final parsing belief vector \( u_t \) describes the model confidence about each of the \( n' \) non-terminals being the root non-terminal. In order for a parse to be successful, the parsing must terminate by reaching a particular non-terminal letter having the role of the start symbol. We allow for the start symbol to emerge in an unsupervised fashion by letting the model learn which of the non-terminals should be considered to have the function of the root of \( G' \). We achieved this by making a constant softmax choice (paying “constant attention”) in the Start Selector Unit: \( o(w), \text{SSU}(u_t) := \text{softmax}(s)^T u_t \), where \( s \) is a trained \( n' \)-dimensional vector. We also experimented with using a two- and three-layer multi-layer perceptron (MLP) networks and found no difference in performance but observed a tendency of the model to encode the productions of the grammar’s start symbol in the MLP, hindering explainability. We also considered fixing one of the entries of \( u_t \) but observed a decrease in recall scores.

E. The Role of clamp(·)

If the hypothesis grammar is or nearly being ambiguous, the vector inner product in NGU often results in belief values in \( u_{i+t} \) being greater than 1 (e.g. as in Fig. 3). To keep the model trainable and explainable at the same time, we limit all grammar unit outputs to 1. The values in our model never go below 0, but we keep the lower bound in the definition of clamp(·) for consistency with similar work in neural networks.

F. Training Losses

The training loss for our model consists of three components, with relative contributions to the total loss being controlled by hyperparameter factors.

1) Prediction loss: The binary cross-entropy between the true labels \((1, 0) \) for positive,negative examples \( w \) and predicted labels \( o(w) \). The prediction loss guides the model towards learning a grammar that generates exactly \( \mathcal{E} \).

2) Sharpening loss: For every entry \( e \) in \( \sigma(P_T), \sigma(P_N^k) \) we compute the sharpening contribution \( S(e) := 1 - (2e - 1)^2 \) and then compute the total sharpening loss \( S := \frac{1}{w(w+1)} \sum_{e} S(e) \). The sharpening loss helps to ensure that the values of \( \sigma(P_T), \sigma(P_N^k) \) are eventually clearly interpretable as productions of \( G' \).

3) Production use loss: Simply the mean of all entries \( e \) of \( \sigma(P_T), \sigma(P_N^k) \), i.e., \( \mathcal{U} := \frac{1}{w(w+1)} \sum_{e} e \). Intuitively, this loss encourages the use of smaller grammars in contrast to larger ones.

4) Total loss: The total loss for a batch \( \mathcal{B} \) is then \( \ell(\mathcal{B}) := \text{BCE}(\mathcal{B}) + \beta S + \gamma \mathcal{U} \), where \( \beta, \gamma \geq 0 \) are hyperparameters.

G. Grammar Extraction

We set a confidence threshold \( \tau \) for when an entry of a production matrix \( P_* \) is to be interpreted as signifying the presence of the production in the grammar. In our experiment, \( \tau = 0.95 \) proved to be a reliable choice, though any...
threshold strictly below 1 is eventually achievable owing to the sharpening loss.

Denote \((M)_{ij}\) the \(i, j\)-th entry of a matrix \(M\), \(a_k\) the \(k\)-th terminal in \(A_T\), and \(a_k\) the \(k\)-th non-terminal in \(A_N\). Then the productions \(\mathcal{P}'\) of the induced grammar \(G'\) can be extracted from the TGU and NGU by the following procedure:

- If \((\sigma(P_T))_{ij} \geq \tau\), add \(A_i \rightarrow t_j\) to \(\mathcal{P}'\).
- For each \(1 \leq k \leq n\), if \((\sigma(P_N^k))_{ij} \geq \tau\) then add \(A_k \rightarrow A_i a_j\) to \(\mathcal{P}'\).
- Let \(\mu := \arg max_k \text{softmax}(s)_k\). Add \(S \rightarrow A_{\mu}\) to \(\mathcal{P}'\).

Note that by constructing \(\mathcal{P}'\) in this manner, \(A_N'\) may contain non-terminals that are never used or that can never be reached from the start symbol \(S\) in a derivation.

H. Parse Tree Construction

While the extracted induced grammar forms a basis for parser construction, the TGU and NGU can be used to construct the parse trees directly. For all \(1 \leq i \leq \ell\), the \(i\)-th clause of the generated grammar \(G'\) is defined as:

\[
\mathcal{P}' = \left\{ \begin{array}{ll}
A \rightarrow a & \quad (\text{terminal productions}) \\
A \rightarrow aA & \quad (A \rightarrow a) \\
A \rightarrow aA & \quad (A \rightarrow A) \\
C \rightarrow bC & \quad (C \rightarrow C) \\
C \rightarrow bC & \quad (C \rightarrow C) \\
C \rightarrow bC & \quad (C \rightarrow C) \\
\end{array} \right.
\]

A. Grammar Generation

During the ground-truth generation phase, we vary the number of terminals \(t\), the number of non-terminals \(n\), and the average number of productions per non-terminal \(p\), thus controlling the complexity of the generated grammar. For each non-terminal \(A_i\), the number of productions is subsequently sampled from a geometric distribution with parameter \(p = 1 - \frac{1}{2}\). Each production for \(A_i\) involves a leaf terminal \(a_j\) with probability \(p_T\) or a non-terminal production of the form \(A_i \rightarrow A_j\) with probability \(p_N = 1 - p_T\). \(A_j\) is chosen uniformly at random from the set of candidate non-terminals and \(a_j\) is chosen uniformly at random from the set of terminals. We designate one symbol in \(A_N\) as the start symbol \(S\) without preference.

In order for the language \(L = \langle G'\rangle\) to be non-empty, we add a randomly sampled terminal production to the set \(\mathcal{P}\) of productions if there are none, and furthermore ensure the derivability reachability of all non-terminals in \(A_N\) from the start non-terminal \(S\) by adding a single production \(A_i \rightarrow A_j a\) for each reachable non-terminal \(A_j\) and some reachable non-terminal \(A_i\), \(A_i, a\) are again taken at random.
B. Generation of Training Examples

Let $\mathcal{G}$ be a ground-truth grammar generated as above, and let $\mathcal{L} = \langle \mathcal{G} \rangle$ be its corresponding language. The training data for a single instance of our model is a dataset $\mathcal{E}$ of words formed from the terminal alphabet $\mathcal{A}_T$, consisting of both negative and positive examples for $\mathcal{L}$. To generate the examples for $\mathcal{L}$, we construct a minimal deterministic finite automaton (DFA) $\mathcal{D}$ equivalent to $\mathcal{G}$ in the sense that $\mathcal{D}$ accepts a word $w$ over $\mathcal{A}_T$ if and only if $w \in \mathcal{L}$.

Constructing finite automata to generate $\mathcal{E}$ instead of using the ground-truth $\mathcal{G}$ directly helps us avoid introducing unintentional bias into $\mathcal{E}$ that would favour $\mathcal{G}$ or grammars very similar to $\mathcal{G}$. In other words, we avoid information leakage by constructing equivalent automata and optimising them prior to generating $\mathcal{E}$.

1) Positive examples: We perform a breadth-first search of depth $d \in \mathcal{D}$, memorising the path taken on each branch of the search. Whenever an accepting state is encountered, the word consisting of transition symbols of the given path read out in sequence is returned. The search then continues to explore the path as before until the depth $d$ has been reached. Observe that the length $\ell$ of the word is $\leq d$.

2) Negative examples from non-accepting paths: Paths (including intermediate paths, i.e. paths of length $\leq d$) of the above breadth-first search that do not end at an accepting state are added to $\mathcal{E}$ as negative examples.

3) Negative examples with invalid prefix: Let $\sigma$ be a state of $\mathcal{D}$ reached by the above search after $k$ steps. Let $\mathcal{A}(\sigma)$ be the set of letters $a \in \mathcal{A}_T$ that label out-transitions of $\sigma$. For all $a \in \mathcal{A}_T \setminus \mathcal{A}(\sigma)$, let $w$ be the word formed by appending $a$ to the $a_1 \ldots a_k$ labels of the transitions on the path traversed to reach $\sigma$. Such $w$ are also negative examples.

4) Negative examples with invalid infix: Let $a_1 \ldots a_k a$ be a word for an invalid suffix negative example, and let $b_1 \ldots b_l$ be the symbols of some valid path between two states of $\mathcal{D}$ such that the latter state is accepting. We consider any word formed as $a_1 \ldots a_k a b_1 \ldots b_l$ a negative example.

5) Random negative examples: To further enrich $\mathcal{E}$, we generated a number of words of uniformly random length up to $d$ with each letter drawn from $\mathcal{A}_T$ uniformly at random such that $\mathcal{D}$ would not accept them, and added them to $\mathcal{E}$ as negatives.

Intuitively, the positive examples guide the model towards learning a grammar $\mathcal{G}'$ that recalls all words of $\mathcal{L}$ (i.e. $\mathcal{L} \subseteq \langle \mathcal{G}' \rangle$), the negative examples with invalid prefix or suffix increase precision (i.e. minimise $\mathcal{L} \cap \langle \mathcal{G}' \rangle$), and the negative examples form non-accepting paths further contribute to increases in precision by discouraging spurious links between the hypothesis non-terminals $\mathcal{A}_N$ of $\mathcal{G}'$ while also indirectly reducing redundancy of computational logic in $\mathcal{G}'$.

We experimented with various ratios of positive to negative examples and ended up settling for 1:1, with random negative examples providing more negative examples wherever needed to reach this ratio.

C. Evaluation

Given a trained model, we extract the induced grammar $\mathcal{G}'$ as per Sect. III-C. We then convert $\mathcal{G}'$ into an equivalent DFA, from whom we compute the minimal DFA $\mathcal{D}'$. The minimality is in the number of states as arrived at by the Hopcroft’s algorithm.

For each regular language $\mathcal{L}$, there exists a unique (up to a re-labelling isomorphism) minimal recognizer DFA $\mathcal{D}$ [11].

---

Fig. 3. An example of parse tree construction from parsing input vectors $v_i$, $u_i$. Expanding on example from Fig. 2 we consider an instance of our model employing grammar $G^* := \langle A_T, A_N, \mathcal{P}^* \rangle$ to parse the word $babc$. $\mathcal{P}^*$ is an extension of $\mathcal{P}'$, with the additional productions introducing ambiguity leading to two more potential parse trees. The ambiguity leads to values exceeding 1 in the dot product in the NGU and clamping is engaged to keep the values between 0 and 1. *Table from the top.* The initial terminal $b$ can be parsed by either of $A$ or $C$. For $i = 2$, each of $A \to Ab, C \to Ab, C \toCb$ can be engaged producing partial parses rooted at $A$ or $C$. For $i = 3$, either $A \to Aa$ or $C \to Ca$ can be used, conditional on the root of the previous intermediate parse. For $i = 4, 5$, $C \to Ca$ applies.

Fig. 4. A visualisation of the training progression of a single run from the experiment described in Sect. IV. The bright and dark tiles represent hypothesis grammar values close to 0 and 1 respectively. From left, we see random hypothesis grammar space becoming more orderly as the training progresses. The final induced grammar after the 15th epoch can be recovered following Sect. III-C.
Given the minimal DFA $D$ of the ground-truth grammar and the minimal DFA $D'$ of the induced grammar, we canonically re-label their states and follow the algorithm of [21] to test whether $D, D'$ are isomorphic. Isomorphism of $D, D'$ means that $G, G'$ are fully equivalent, implying that our model has achieved the maximum recall and precision and that we do not need to evaluate further.

In the case that $G, G'$ are not equivalent, we assess similarity of languages by comparing the finite subsets $L_d, L'_d$ of each. These consist of all words of languages by comparing the finite subsets $L_d, L'_d$ of each. We then measure the recall $\frac{|L_d \cap L'_d|}{|L_d|}$, precision $\frac{|L_d \cap L'_d|}{|L'_d|}$, and accuracy $\frac{|L_d \cap L'_d|}{|L_d \cup L'_d|}$ of our model.

**D. Results**

We generated grammars with $t = 4$, $n \in \{2, 3, 4\}$, and $p \in \{2, 3, 4, 5\}$, resulting in 12 different grammar configurations. For each grammar, we generated the full dataset of positive and negative examples of strings up to length 16, training models on strings of length up to 6, 8, 10, 12, 14, and 16, with 5 runs per grammar per length. During training, we used a batch size of 80. Adam optimizer with learning rate 0.005, $\beta$ of 0.95 after 60% of epochs and 0 before then. There was a maximum of 60 epochs and $n^t = 5$. We extracted the grammar from the model using a confidence threshold $\tau = 0.95$, and evaluated it as described above.

Overall, the model learned the grammar from which the data was generated exactly (i.e. the automata $D, D'$ were isomorphic) in 85% of all runs.

For more complex grammars, we observed an increase in accuracy (+2-5%) when increasing length of examples while keeping the grammar complexity fixed. Generally, grammars with smaller numbers of non-terminals $n$ were easier to learn for our model, and for all $n$, higher number of productions per non-terminals also led to better results. This is somewhat counter-intuitive but perhaps due to our methodology for negative example generation: more complex grammars had more negative examples from non-accepting paths and fewer random negative examples.

As pointed out in Sect. [22] no previous method addressed the problem of learning the entire class of regular grammars from positive and negative examples, and so no direct comparison can be made at present. Nevertheless, our model and detailed results are made open to further evaluation and extensions, with reproducibility and hope for aiding future research in mind.

**V. Conclusion**

We have introduced a purely neural explainable model for the induction of regular grammars from positive and negative acceptor examples, and demonstrated its ability to induce grammars to high levels of accuracy.

Our model can be used both for grammar induction and as a regular language parser. This duality of purpose arises from it simultaneously inducing a grammar and attempting partial parses throughout its training as an acceptor automaton.

The ultimate but distant goal of grammatical inference in the context of algorithm learning is the induction of general grammars (TYPE-0 in the Chomsky hierarchy) which are equivalent in power to Turing machines. We see regular grammars as a simple but rich class of grammars for induction from examples under the Programming by Example paradigm and hope that our work will help to facilitate further advancements in explainable neural grammar inference. To this end, we make our code and data available online.

**REFERENCES**


