A Neural Model for Regular Grammar Induction

Anonymous submission for double-blind review

***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. We show that our method consistently attains high recall and precision scores across a range of tests of varying complexity, and make it readily available[1] and open to extensions to broader classes of grammars.**

***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 to simpler devices entirely characterisable by mappings of individual inputs to their corresponding outputs, finite automata may need the entire input history to determine their next state or an output.

The learning of finite automata is a classical problem in computational learning theory [21], and can be phrased as follows: given a language $\mathcal{L}$ and a set of examples $\mathcal{E}$, use $\mathcal{E}$ to find a finite acceptor automaton $\mathcal{D}$ that accepts words in $\mathcal{L}$ and rejects words not in $\mathcal{L}$, or at least find a $\mathcal{D}$ that performs these duties with sufficient accuracy. For instance, $\mathcal{L}$ can be the language generated by the regular expression a^*bb^*. The set of examples $\mathcal{E}$ could consist of the entire $\mathcal{L}$, 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, $\mathcal{E}$ could contain words such as aab and bbb marked as positive examples (the words $\mathcal{D}$ should accept), and aaa and aba marked as negative examples (words to be rejected by $\mathcal{D}$).

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 $\mathcal{E}$, while others preferred automata that generalised well to $\mathcal{L}$ in scenarios where $\mathcal{E}$ contains only a few examples representative of the original language. Another degree of freedom arises in the constraints that are placed on $\mathcal{E}$. One can insist that $\mathcal{E}$ consists of only positive examples (i.e. $\mathcal{E} \subseteq \mathcal{L}$), that $\mathcal{E}$ contains enough examples to sanction good generalisation to $\mathcal{L}$ (e.g. all repetitions of a pattern beyond a certain count signify that the pattern may be repeated indefinitely), or that $\mathcal{E}$ contains both positive and negative examples (i.e. $\mathcal{E} \cap \mathcal{L} \neq \emptyset \neq \mathcal{E} \cap \mathcal{L}^c$, where $\mathcal{L}^c$ denotes words not in $\mathcal{L}$). 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 $\mathcal{L}$ from a given $\mathcal{E}$ 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 $\mathcal{L}$ despite being based on only $\mathcal{E}$, 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 $\mathcal{G}$ is a triplet of $\mathcal{A}_T, \mathcal{A}_N, \mathcal{P}$ – the terminal alphabet, non-terminal alphabet, and set of productions, respectively, where one of the letters of $\mathcal{A}_N$ is marked as the start symbol (the "root" of $\mathcal{G}$). A left-regular grammar is a grammar in which all productions of $\mathcal{P}$ are of the form $A \to c$, $A \to \epsilon$, or $A \to Bc$ for $A, B \in \mathcal{A}_N$, $c \in \mathcal{A}_T$, ϵ the empty string. A right-regular grammar uses productions of the form $A \to cB$ instead of $A \to Bc$. Regular grammars are equivalent to finite automata in their language-generating power [11] and admit straightforward conversions into each other [20]. We do not consider empty languages or grammars that give empty languages.

In our setting, there is a target language $\mathcal{L} = \langle \mathcal{G} \rangle$ generated by the ground-truth grammar $\mathcal{G}$. We are shown examples $\mathcal{E}$ taken from both $\mathcal{L}$ and $\mathcal{L}^c$, and our model internally learns a hypothesis grammar $\mathcal{G}'$. In other words, we do not train neural automata to become accurate acceptors for given languages $\mathcal{L}$, 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 weak 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.
- 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.
- Finally, we present the conclusions of a loss ablation study assessing the impact of our training losses on the accuracy and conciseness of the induced grammars.

[1] We make our code available at anonymised.

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 [28], [31] and probabilistic finite state machines [7] 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 [5] and for learning of unambiguous regular expressions of maximum loop depth 2 [23]. A genetic programming approach leveraging both positive and negative examples was used in [32], and a further polynomial-time method for 1-unambiguous regular expressions aiming for simplicity of the resulting expressions was presented in [15]. 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 [3]. [27], [36] focus on identifying email campaigns with high precision while phrasing it as an optimisation problem, and [6] 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.

The current research in this area foregoes the notion of finite automata altogether and employs recurrent [26], [33] and transformer [14], [25], [34] neural architectures for nearly all of the tasks previously addressed by learning regular expressions. These models are deep and in general not explainable, though some recent work has begun to address this issue [2].

Grammar induction. Pioneering work on the learning of grammars from examples often attempted to construct probabilistic context-free grammars that generated the target language $\mathcal{L}$ [16]. Subsequent attempts employed a wider variety of tactics including Bayesian methods [9], hill-climbing (reward or quality maximisation) [13], and genetic programming [35]. The learning was done chiefly from positive examples, though methods for automatic negative example generation from positive examples were later introduced [30]. 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 [4], [10], [24], reinforcement learning models [37], and partially interpretable structured attention [22] and transformer [17] 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 [29]. 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 [12] 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 $w{:}a_1a_2a_3\dots a_\ell$ we one-hot encode letters a_i into t-dimensional vectors v_i, where $t = |\mathcal{A}_T|$ 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 n'-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 [8], [18].

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_ℓ 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

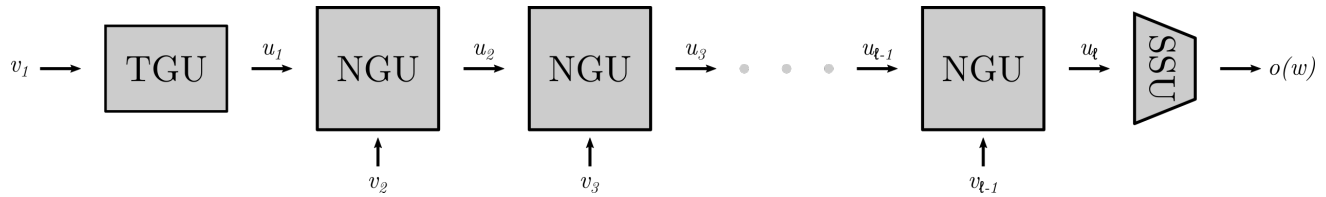

Fig. 1. An overview of the model's neural structure. *From left.* The Terminal Grammar Unit is applied to the first letter of the input word w. The non-terminal grammar unit is then applied recurrently (i.e. the trained weights are shared between individual instances) to the remaining letters. The final parse belief vector u_ℓ is fed into the Start Selector Unit to yield the accept/reject verdict.

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:

$$\text{TGU}\left(v_1\right) := \text{clamp}\left(\sigma\left(P_T\right) v_1\right),$$

where $\text{clamp}(v) := \max\left(\min\left(1, v\right), 0\right)$ and σ 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 $\sigma\left(P_T\right)$ represents the belief of the model (the strength of TGU's hypothesis) that the production $A \to b$ is a part of the grammar $\mathcal{G}'$ being learned from $\mathcal{E}$.

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 $\mathcal{A}'_N$ on the left-hand side. To perform its parse, the NGU takes the previous parse vector u_i, current terminal vector v_{i+1}, and yields u_{i+1} where the k-th entry of u_{i+1} for $1 \leq k \leq n'$ is given by

$$\text{NGU}\left(u_i, v_{i+1}\right)_k := \text{clamp}\left(\left(\sigma\left(P_N^k\right) v_{i+1}\right)^{\mathrm{T}} u_i\right).$$

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 $\sigma\left(P_N^k\right)$ represents the strength of NGU's hypothesis that the production $A \to Bc$ belongs to $\mathcal{P}'$.

D. Start Selector Unit (SSU)

The final parsing belief vector u_ℓ 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 $\mathcal{G}'$. We achieved this by making a constant softmax choice (paying "constant attention") in the Start Selector Unit:

$$o(w), \text{SSU}\left(u_\ell\right) := \text{softmax}\left(s\right)^{\mathrm{T}} u_\ell,$$

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_ℓ but observed a decrease in recall scores.

E. The Role of clamp$(\cdot)$

If the hypothesis grammar is or nears being ambiguous, the vector inner product in NGU often results in belief values in u_{i+1} 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$(\cdot)$ for consistency with similar work in neural networks.

In our experimentation, we found that clamping together with an appropriate loss encouraging the use of fewer productions was often very effective at reducing ambiguity, even eliminating it altogether.

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 $o(w)$

$$\text{BCE}\left(y, o(w)\right) := y \log\left(o(w)\right) + \left(1 - y\right) \log\left(1 - o(w)\right)$$

The prediction loss guides the model towards learning a grammar that generates exactly $\mathcal{L}$.

2) Sharpening loss: For every entry e in $\sigma\left(P_T\right), \sigma\left(P_N^k\right)$ we compute the sharpening contribution

$$\mathcal{S}\left(e\right) := \left(1 - \left(2e - 1\right)^2\right)$$

and then compute the total sharpening loss

$$\mathcal{S} := \frac{1}{n'(n'+1)t} \sum_e \mathcal{S}\left(e\right).$$

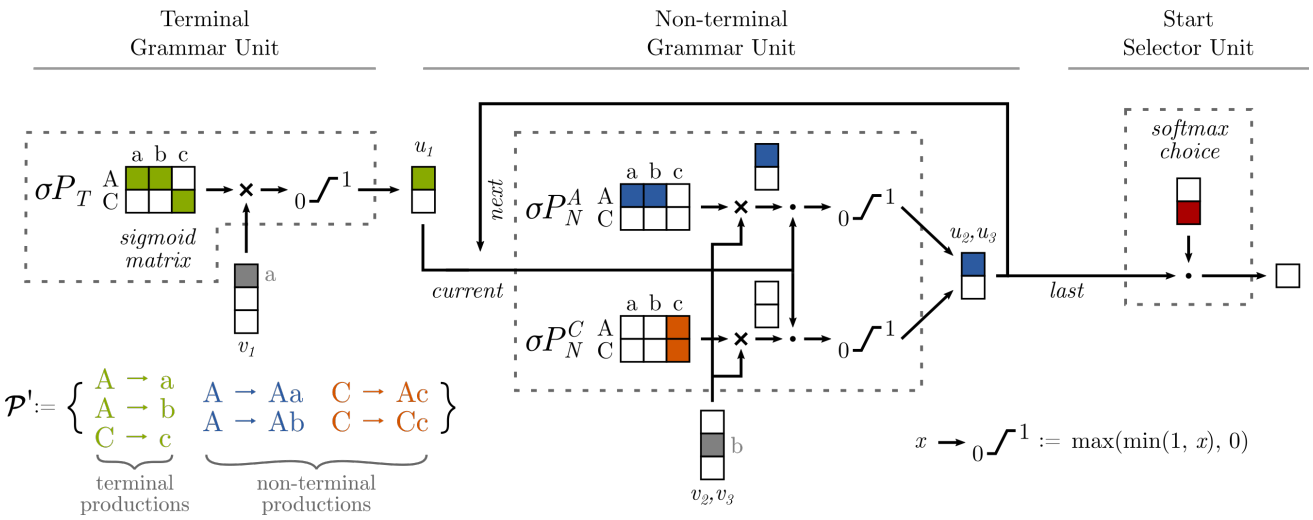

Fig. 2. An illustration of the model's internals. The productions $\mathcal{P}'$ of the model hypothesis grammar $\mathcal{G}' = (\mathcal{A}_T, \mathcal{A}'_N, \mathcal{P}')$ generate the language given by the regular expression $(a|b)^* cc^*$. The $\times$ symbol represents matrix multiplication, $\cdot$ represents vector inner product, and the colour coding of productions and matrix entries marks equivalences by the encoding-decoding procedures of Sect. III-C and Sect. III-G. We present the model with a string *abb*. *Left.* The Terminal Grammar Unit matches the initial a encoded as $v_1 = (1\,0\,0)^{\mathrm{T}}$ with the production $A \to a$ and produces the belief vector $u_1 = (1\,0)^{\mathrm{T}}$. *Middle.* The Non-terminal Grammar Unit has its production matrices queried by $v_2 = (0\,1\,0)^{\mathrm{T}}$ and finds a match in the production $A \to Ab$. This is then dotted with the prior belief u_1 to produce the next parse belief u_2. The same is repeated for $v_3, u_2 = v_2, u_1$ giving $u_\ell = u_3 = (1\,0)^{\mathrm{T}}$. *Right.* The model has been trained to recognize C as the root symbol, but the terminal parse u_ℓ on the given negative example is $(1\,0)^{\mathrm{T}}$, leading to 0 as the output of the acceptor.

The sharpening loss helps to ensure that the values of $\sigma(P_T), \sigma(P_N^k)$ are eventually clearly interpretable as productions of $\mathcal{G}'$.

3) Production use loss: Simply the mean of all entries e of $\sigma(P_T), \sigma(P_N^k)$, i.e.

$$\mathcal{U} := \frac{1}{n'(n'+1)t} \sum_e e\,.$$

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

$$\ell(\mathcal{B}) := \mathrm{BCE}(\mathcal{B}) + \beta\mathcal{S} + \gamma\mathcal{U},$$

where $\beta, \gamma \geq 0$ are hyperparameters.

G. Grammar Extraction

We set a confidence threshold τ for when an entry of a production matrix $P_\bullet$ is to be interpreted as signifying the presence of the production in the grammar. In our experimentation, $\tau = 0.95$ 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 $\mathcal{A}_T$, and A_k the k-th non-terminal in $\mathcal{A}'_N$. Then the productions $\mathcal{P}'$ of the induced grammar $\mathcal{G}'$ can be extracted from the TGU and NGU by the following procedure:

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

Note that by constructing $\mathcal{P}'$ in this manner, $\mathcal{A}'_N$ 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 $1 \leq i \leq \ell$, u_ℓ multi-hot-encodes all non-terminals that can be used to generate the prefix sub-word $w_{\leq i} = a_1 \ldots a_i$ of w. This can be done inductively in i as follows: for $1 \leq j \leq n$, if $\left(\sigma(P^k)v_i\right)_j \geq \tau$ and $u_i \geq \tau$, N_k is a parse tree node for u_{i+1} with one child being the root of the parse tree for u_i and the other being the leaf terminal a_i. This is illustrated in Fig. 3. Such a tree may exist for every j, giving all the at most n' possible parse tree roots for any prefix sub-word $w_{\leq i}$.

IV. Experiments

To systematically evaluate the grammar induction performance of our model, we generate left-regular grammars of varying complexity using a randomized procedure, and then use the said grammars to produce training datasets consisting of positive and negative examples. Once trained, the instances of our model are inspected for their induced grammars $\mathcal{G}'$ and tested for how closely they match the ground-truth $\mathcal{G}$.

Due to the lack of related work addressing the problem of general regular grammar induction from positive and negative examples (cf. Sect. II), the objective of our experimentation is to investigate the robustness of our method to increases in grammar complexity and potential brevity in training examples (i.e. the cases when the training examples may be plenty but are not long enough to be wholly confident about the

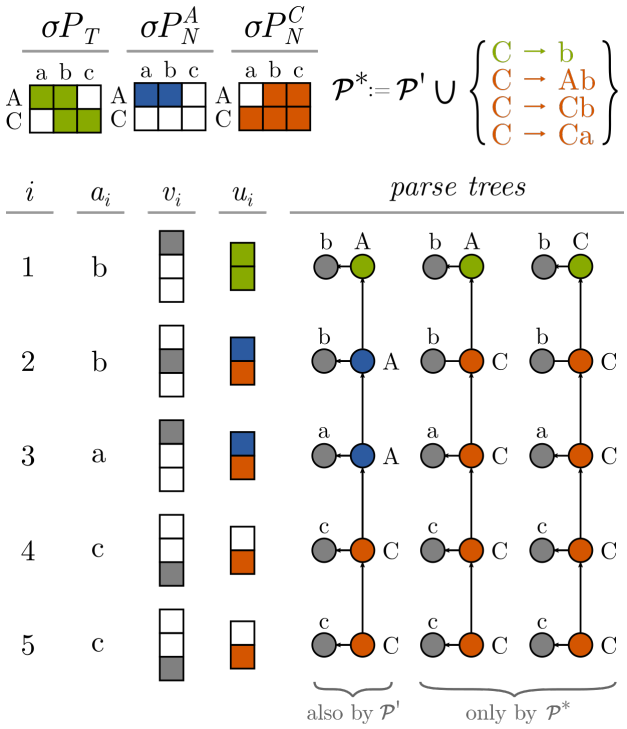

Fig. 3. An example of parse tree construction from parsing input vectors v_i, u_i. Expanding on example from Fig. 2 we consider an instance of our model employing grammar $\mathcal{G}^* := (\mathcal{A}_T, \mathcal{A}'_N, \mathcal{P}^*)$ to parse the word *bbacc*. $\mathcal{P}^*$ is an extension of $\mathcal{P}'$, 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 $A \to Ab, C \to Ab, C \to Cb$ 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 $i = 4, 5$, $C \to Cc$ applies.

recurrence in a production). This is in line with the example evaluation approach taken in other work [3], [27], [32], [36].

A. Grammar Generation

During the ground-truth grammar generation phase, we vary the number of terminals t, the number of non-terminals n, and the average number of productions per non-terminal $\mathfrak{p}$, 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 $\pi = \mathfrak{p}^{-1}$. Each production for A_i is either a terminal production of the form $A_i \to a$ with probability $p_T = 0.4$ or a non-terminal production of the form $A_i \to A_j a$ with probability $p_N = 1 - p_T$. A_j is chosen uniformly at random from the set $\mathcal{A}_N$ of candidate non-terminals and a is chosen uniformly at random from the set $\mathcal{A}_T$ of terminals. We designate one symbol in $\mathcal{A}_N$ as the start symbol S without preference.

In order for the language $\mathcal{L} = \langle \mathcal{G} \rangle$ 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 derivation reachability of all non-terminals in $\mathcal{A}_N$ from the start non-terminal S by adding a single production $A_i \to A_j a$ for each unreachable non-terminal A_j and some reachable non-terminal A_i. A_i, a are again taken at random.

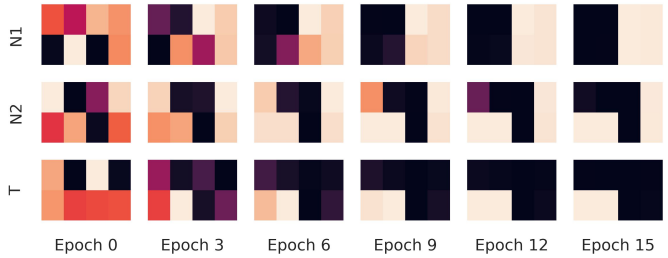

Fig. 4. A visualisation of the training progression of a single run from the experiment described in Sect. IV. The bright and dark tiles represent hypothesis grammar values close to 0 and 1 respectively. From left, we see random hypothesis grammar space becoming more orderly as the training progresses. The final induced grammar after the 15th epoch can be recovered following Sect. III-G to give $\mathcal{G}' = (\{a, b, c, d\}, \{N_1, N_2\}, \mathcal{P}')$, where $\mathcal{P}' = \{N_2 \to a, N_2 \to B, N_1 \to N_1 c, N_1 \to N_1 d, N_1 \to N_2 c, N_1 \to N_2 d\} \cup \{N_2 \to N_1 d, N_2 \to N_1 a, N_2 \to N_1 b, N_2 \to N_1 d\}$.

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 on $\mathcal{D}$, 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 ℓ 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 postfix: Let σ 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 σ. For all $a \in \mathcal{A}_T \backslash \mathcal{A}(\sigma)$, let w be the word formed by appending a to the $a_1 \dots a_k$ labels of the transitions on the path traversed to reach σ. Such w are also negative examples.

4) Negative examples with invalid infix: Let $a_1 \dots a_k a$ be a word for an invalid postfix negative example, and let $b_1 \dots b_l$ be the symbols of some valid path between two states of $\mathcal{D}$

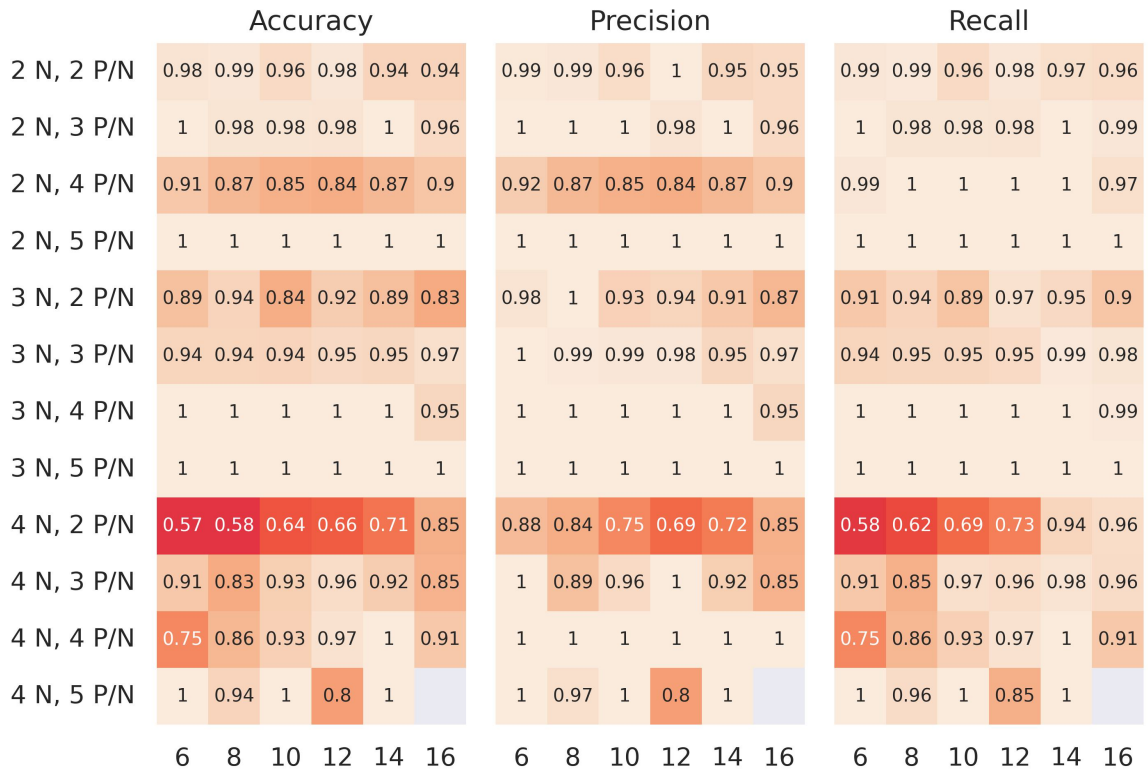

Fig. 5. The results of the experiments. *Horizontally.* The length up to which training examples were used in the corresponding runs. *Vertically.* The number of non-terminal and the average productions per non-terminal of the grammar $\mathfrak{p}$, giving an intuition on its complexity. The experiment for four non-terminals, 5 productions per non-terminal, and training examples drawn from strings up to length 16 was not run due to the limited memory available to our training device.

such that the latter state is accepting. We consider any word formed as $a_1 \dots a_k a b_1 \dots b_l$ 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 infix increase precision (i.e. minimise $\mathcal{L}^c \cap \langle \mathcal{G}' \rangle$), 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-G. 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 Hopcroft's algorithm [19].

For each regular language $\mathcal{L}$, there exists a unique (up to a re-labelling isomorphism) minimal recognizer DFA $\mathcal{D}$ [20, p. 159-164]. Given the minimal DFA $\mathcal{D}$ of the ground-truth grammar and the minimal DFA $\mathcal{D}'$ of the induced grammar, we canonically re-label their states and follow the algorithm of [1] to test whether $\mathcal{D}, \mathcal{D}'$ are isomorphic. Isomorphism of $\mathcal{D}, \mathcal{D}'$ means that $\mathcal{G}, \mathcal{G}'$ 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 $\mathcal{G}, \mathcal{G}'$ are not equivalent, we assess similarity of languages by comparing the finite subsets $\mathcal{L}_d, \mathcal{L}'_d$ of each. These consist of all words of $\mathcal{L}, \mathcal{L}' = \langle \mathcal{G} \rangle$ up to length d. We then measure the recall $\frac{\mathcal{L}_d \cap \mathcal{L}'_d}{\mathcal{L}_d}$, precision $\frac{\mathcal{L}_d \cap \mathcal{L}'_d}{\mathcal{L}'_d}$, and accuracy $\frac{\mathcal{L}_d \cap \mathcal{L}'_d}{\mathcal{L}_d \cup \mathcal{L}'_d}$ of our model.

D. Results

We generated grammars with $t = 4$, $n \in \{2, 3, 4\}$, and $\mathfrak{p} \in \{2, 3, 4, 5\}$, 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, β of 0.05 after 60% of epochs and 0 before then. There was a maximum of 60 epochs and $n' = 5$. We show an example

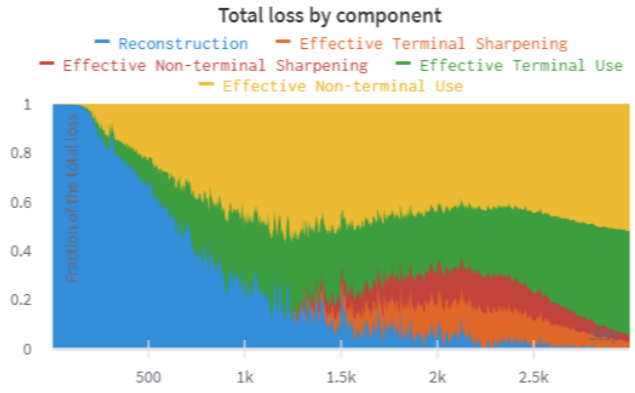

Fig. 6. The breakdown of the total training loss by component. We observe that at the beginning of the training, the reconstruction loss dominates all other components. Terminal and non-terminal use slowly grow to approximately mid-training, signifying the storage of production information within the model. Then, the sharpening losses are engaged. In the end, the entire training loss is composed of terminal and non-terminal use losses.

training progress in terms of individual loss components in Fig. 6. We extracted the grammar from the model using a confidence threshold $\tau = 0.95$, and evaluated it as described above.

The results are shown in Fig. 5. Overall, the model learned the grammar from which the data was generated *exactly* (i.e. the automata $\mathcal{D}, \mathcal{D}'$ were isomorphic) in 85% of all runs.

For more complex grammars, we observed an increase in accuracy when increasing length of examples while keeping the grammar complexity fixed. This was most clear in grammars with 4 non-terminals and 2 productions per non-terminal. The last row of Fig. 5 required the biggest datasets. Due to the limited memory available on our training hardware only comparatively few runs (2-3) have been concluded per entry (as memory swapping was time-consuming). Note, however, that such problems can be circumvented by providing the training procedure with examples on-demand (rather than generating them all in advance as was the case in our setup).

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.

It would be our sincere wish to compare this performance of our model to one of the existing methods. However, as pointed out in Sect. II, 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 is made open[1] to further evaluation and extensions with reproducibility and hope for aiding future research in mind.

E. Experimental Loss Ablations

Similarly to the dual role of negative examples from non-accepting paths, the "grammar quality" losses $\mathcal{U}, \mathcal{S}$ (cf. Sect. III-F) affect both the performance of our model and the quality of the induced grammars, sometimes even cancelling each other out to large extent. We have made the following observations when fine-tuning our models:

- Engaging $\mathcal{S}$ too early into the training leads to a noticeable decrease in model's recall. This is in agreement with our interpretation of $\mathcal{S}$ as of a loss that encourages decisions on the presence or absence of individual productions. Doing so too early forces the model to decide before it has seen enough training data to make a decision founded on evidence from examples. As a consequence, most of our experiments increased β only very slowly and after the training has been sufficiently advanced.
- Applying $\mathcal{U}$ from the beginning (with small values of γ) lead to notably higher accuracy scores at little cost in recall.
- The loss configuration of approximately $3\beta = 2\gamma$, when both were engaged right from the beginning of the training, had the tendency to extend the training time and lead to more unsettled behaviour of the production matrices towards the end of the training. As above, we therefore tended to keep γ very small from the beginning and allow β to climb only once the training had progressed sufficiently and that to a maximum an order of magnitude larger than γ.

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

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. Testing the equivalence of regular languages. *arXiv preprint arXiv:0907.5058*, 2009.

[2] Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Igor Khmelnitsky, Martin Leucker, Daniel Neider, Rajarshi Roy, and Lina Ye. Extracting context-free grammars from recurrent neural networks using tree-automata learning and a* search. In *International Conference on Grammatical Inference*, pages 113–129. PMLR, 2021.

[3] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. *Computer*, 47(12):72–80, 2014.

[4] Samuel R Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. *arXiv preprint arXiv:1603.06021*, 2016.

[5] Alvis Brāzma. Efficient identification of regular expressions from representative examples. In *Proceedings of the sixth annual conference on Computational learning theory*, pages 236–242, 1993.

[6] Duy Duc An Bui and Qing Zeng-Treitler. Learning regular expressions for clinical text classification. *Journal of the American Medical Informatics Association*, 21(5):850–857, 2014.

[7] Rafael C Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference*, pages 139–152. Springer, 1994.

[8] Ricky TQ Chen, Xuechen Li, Roger B Grosse, and David K Duvenaud. Isolating sources of disentanglement in variational autoencoders. *Advances in neural information processing systems*, 31, 2018.

[9] Stanley F Chen. Bayesian grammar induction for language modeling. *arXiv preprint cmp-lg/9504034*, 1995.

[10] Jihun Choi, Kang Min Yoo, and Sang-goo Lee. Learning to compose task-specific tree structures. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[11] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.

[12] John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, 1969.

[13] Craig M Cook, Azriel Rosenfeld, and Alan R Aronson. Grammatical inference by hill climbing. *Information Sciences*, 10(1):59–80, 1976.

[14] Ehsan Doostmohammadi, Minoo Nassajian, and Adel Rahimi. Persian ezafe recognition using transformers and its role in part-of-speech tagging. *arXiv preprint arXiv:2009.09474*, 2020.

[15] Henning Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.

[16] King-Sun Fu and Taylor L Booth. Grammatical inference: Introduction and survey-part ii. *IEEE Transactions on Systems, Man, and Cybernetics*, (4):409–423, 1975.

[17] Ben Goertzel, Andrés Suárez-Madrigal, and Gino Yu. Guiding symbolic natural language grammar induction via transformer-based sequence probabilities. In *International Conference on Artificial General Intelligence*, pages 153–163. Springer, 2020.

[18] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. 2016.

[19] John E Hopcroft. *A linear algorithm for testing equivalence of finite automata*, volume 114. Defense Technical Information Center, 1971.

[20] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[21] Michael J Kearns and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[22] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M Rush. Structured attention networks. *arXiv preprint arXiv:1702.00887*, 2017.

[23] Efim Kinber. Learning regular expressions from representative examples and membership queries. In *International Colloquium on Grammatical Inference*, pages 94–108. Springer, 2010.

[24] Jean Maillard, Stephen Clark, and Dani Yogatama. Jointly learning sentence embeddings and syntax with unsupervised tree-lstms. *Natural Language Engineering*, 25(4):433–449, 2019.

[25] Artem A Maksutov, Vladimir I Zamyatovskiy, Viacheslav O Morozov, and Sviatoslav O Dmitriev. The transformer neural network architecture for part-of-speech tagging. In *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, pages 536–540. IEEE, 2021.

[26] Juan Antonio Perez-Ortiz and Mikel L Forcada. Part-of-speech tagging with recurrent neural networks. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, volume 3, pages 1588–1592. IEEE, 2001.

[27] Paul Prasse, Christoph Sawade, Niels Landwehr, and Tobias Scheffer. Learning to identify regular expressions that describe email campaigns. *arXiv preprint arXiv:1206.4637*, 2012.

[28] Lawrence Rabiner and Biinghwang Juang. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

[29] David Canfield Smith, Allen Cypher, and Larry Tesler. Programming by example: novice programming comes of age. *Communications of the ACM*, 43(3):75–81, 2000.

[30] Noah A Smith and Jason Eisner. Guiding unsupervised grammar induction using contrastive estimation. In *Proc. of IJCAI Workshop on Grammatical Inference Applications*, pages 73–82, 2005.

[31] Andreas Stolcke and Stephen Omohundro. Inducing probabilistic grammars by bayesian model merging. In *International Colloquium on Grammatical Inference*, pages 106–118. Springer, 1994.

[32] Borge Svingen. Learning regular languages using genetic programming. In *Proc. 3-rd Genetic Programming Conference*, pages 374–376, 1998.

[33] Peilu Wang, Yao Qian, Frank K Soong, Lei He, and Hai Zhao. Part-of-speech tagging with bidirectional long short-term memory recurrent neural network. *arXiv preprint arXiv:1510.06168*, 2015.

[34] Shijie Wu, Ryan Cotterell, and Mans Hulden. Applying the transformer to character-level transduction. *arXiv preprint arXiv:2005.10213*, 2020.

[35] Peter Wyard. Context free grammar induction using genetic algorithms. In *IEE colloquium on grammatical inference: theory, applications and alternatives*, pages P11–1. IET, 1993.

[36] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, Geoff Hulten, and Ivan Osipkov. Spamming botnets: signatures and characteristics. *ACM SIGCOMM Computer Communication Review*, 38(4):171–182, 2008.

[37] Dani Yogatama, Phil Blunsom, Chris Dyer, Edward Grefenstette, and Wang Ling. Learning to compose words into sentences with reinforcement learning. *arXiv preprint arXiv:1611.09100*, 2016.