Neuralising CYK Parser

Jannis Widmer Qi Wu Oliver Sihlovec Luca Hollenstein

Abstract

The CYK algorithm can decide whether a given string of any length is in a given context-free grammar or not. However, its use cases are limited, because the production grammar must be available as an input parameter, hence the algorithm being inapplicable in scenarios where the context-free grammar is unknown. In this work, we build a neuralized version of the classical CYK algorithm, such that it can learn and generalize to more softly defined predictions. To that end, we build a neuralized version of the classical CYK Parser, which can learn purely by examples and can be trained end-to-end in a supervised setting.

1. Introduction

NLP was largely dominated by context-free grammar in the past, initially described directly by humans, later by learning the rules based on large datasets (Zanzotto et al., 2020). More recently deep learning based methods have entered the stage, specifically recurrent networks (especially LSTM-based models) and Transformers. Both approaches try to overcome the context-free assumption, making predictions as contextualized as possible (Zanzotto et al., 2020). While theoretical analysis shows that both LSTM and Transformer-based models are in principle Turing-complete, it has also been shown that in practice those models often fail to even imitate simple machines (Shi et al., 2022). This opens a window for different, more optimal approaches to the problem.

In this paper we design a "neuralized" implementation of the CYK Parser, that is an implementation of the CYK Parser, that only uses differentiable expressions and is hence end-to-end trainable using gradient descent, while still being as closely related as possible to the original version. We evaluate the implementation in terms of classification accuracy on several synthetic datasets of context-free languages, comparing it with the results obtained using LSTM and Transformer. Note that all our models and datasets are available at github[1].

To our best knowledge, there has been no previous research for exactly this specific topic. There have, however, been similar research directions, for instance Zanzotto et al. (Zanzotto et al., 2020) have developed a version of the CYK Parser that approximates the original CYK Parser using a distributed representation. As far as we understand, their algorithm is however not directly applicable to our use case as it is still required to know the grammar in advance. Chua et al.(Chua & Duffy, 2021) have developed DeepCPCFG, a deep learning based system for extracting information from documents into relational databases. The network they use for the extraction into a tree structure has in fact large similarities to our proposed network. Their work differs from ours in that their main focus is to train the network to predict good parse trees using valid parse trees as ground truth data.

Our contributions are:

- The development of a model for the task of predicting whether a sequence belongs to a context-free language or not, surpassing LSTM and Transformer based models in most cases. Our model is especially robust with respect to out-of-distribution data.
- Just like Schlag et al.(Schlag & Schmidhuber, 2021) we showcase how a classic algorithm can be "neuralized", such that it becomes trainable. Our results indicate that those neuralized algorithms then tend to work very well in their original domain.

2. Models and Methods

2.1. CYK Parser

2.1.1. Classic CYK Parser

To that end, we first present the classical CYK Parser. This is a dynamic programming algorithm, that can determine whether a given string is in a language described by context-free grammar in Chomsky normal form (CNF). Via backtracking, it can also derive a valid parse tree if the string is indeed part of the language. It should be noted that every context-free grammar can be converted into CNF (Chomsky, 1959), so a CYK Parser can indeed parse every context-free language. A grammar is in CNF if all of its rules are of the form $A \to BC$ or $A \to a$ or $S \to \varepsilon$, where a is a terminal symbol, ε is the empty string, A, B, C are non-terminal symbols and S is the start symbol. Moreover, B nor C may be the start symbol (D'Antoni).

[1] https://github.com/jnice-81/DLCykParser

Algorithm 1 CYK Parser

```
Input: string s, grammar g
if s already computed then
    return computed
end if
result := ∅
if s has length one then
    result := set of rules that can generate s
else
    for u, v possible split of s do
        r1 := call CYK parser on u
        r2 := call CYK parser on v
        r3 := r1 × r2
        r4 := Set of rules that produce an element in r3
        result := result ∪ r4
    end for
end if
store result together with s
return result
```

Algorithm 2 Neuronal CYK Parser

```
Input: string s
if s already computed then
    return computed
end if
result := vector of −∞ ∈ R^d
if s has length one then
    result := embedding for s
else
    for u, v possible split of s do
        r1 := call neuronal CYK parser on u
        r2 := call neuronal CYK parser on v
        r3 := concatenate r1 and r2
        r4 := Residual network on r3
        r5 := linear layer in R^{d×2∗d} on r4
        r6 := Residual network on r5
        result := elementwiseMax(result, r6)
    end for
end if
store result together with s
return result
```

The algorithm is shown in algorithm1. After the algorithm terminates, it is possible to determine whether s is part of the language by examining whether the returned result set contains the start symbol of the grammar. Note that we changed the presentation of the algorithm from a more classical presentation as a DP algorithm to a memoization-based algorithm, as we consider the latter easier to understand. The algorithm runs in $O(n^3)$, where n is the length of the string, and requires $O(n^2)$ memory. (Graham & Harrison, 1976)

2.1.2. Neuralized CYK Parser

We create a neuralized version of the classic CYK parser by replacing non-differentiable operations in the classic algorithm with differentiable ones. This then enables us to train the algorithm using gradient descent. The used algorithm is shown in algorithm 2.

We introduce embedding vectors in $\mathbf{R}^d$ for all terminal symbols of the grammar. d should be the number of rules in the grammar. Note that in practice using a broad approximation for the number of rules also works. We use a final linear layer mapping from $\mathbf{R}^d$ to 2, and use cross entropy loss to train the network.

The essential idea for the algorithm is as follows: We replaced the set of string-generating rules in the original CYK parser with a vector with dimensionality equal to the total number of rules in the grammar. For the training however, we assumed a tunable hyperparameter to at least resemble a black box setting. It's easy to see that this vector is in principle complex enough to store the entire set of generating rules for a string - one could for example use one-hot encoding on the vector to indicate whether the n-th rule can generate the given string.

The product rule part in the original CYK Parser is replaced by a residual MLP mapping from the two vectors describing the strings to a vector describing the concatenation of both. (Again of dimensionality d) It is a residual network to avoid vanishing gradients, as our network depth grows with the size of the input string.

It should be noted that while we use embeddings for the terminal symbols, those vectors could in principle also be generated differently. For example, one could easily use a CNN to extract a vector describing the symbol e.g. from MNIST images.

Our model inherits the complexity properties, in terms of sequence length, of the original CYK parser for the most part: The runtime is in $O(n^3)$ and the memory consumption in $O(n^2)$. During training however, the memory rises to $O(n^3)$, as we need to store gradients for backpropagation.

2.1.3. Outer product CYK

At the beginning of this work, we intended to replace the product rule part instead with $r := (max(W_1 * u@v^T), ..., max(W_d * u@v^T))^T$, where u and v are the vectors describing the two strings being concatenated, @ is a matrix multiplication, $*$ is an elementwise multiplication and W_i is a trainable weight matrix in $\mathbf{R}^{d \times d}$.

It should be visible that this is essentially a direct reimplementation of the original CYK parser in terms of mathe-

matical operations - if one knows the grammar in advance, then it would be possible to initialize the embeddings for a symbol to 1 if the i_{th} rule can produce this symbol 0 otherwise, and initialize $W_{i,kz}$ to 1 if the i_{th} rule can produce a concatenation of the k_{th} and the z_{th} rule. The algorithm will then always output 1 if a string is described by the grammar and 0 otherwise.

While this works for a given grammar, in practice it fails to converge when trained from examples, except for very simple grammar.

2.2. Standard Neural Network Baseline

To test the performance in terms of accuracy of the NCYK Parser, we compare our model against two standard neural networks, LSTM and Transformer, and train them to solve the same binary classification problem, whether a sequence of symbols is created from a fixed context-free grammar. In our case, the input sequence's symbols are characters, so we use a dictionary to convert each character to an index. This encoding can be expanded further for any set of symbols, e.g. natural languages, etc. Furthermore, we pad the sequence to the length of the longest sequence with zeros. The parameters of the models can be seen in the Appendix: A.1

2.3. Datasets

To test our suggested architecture, it is important to have unbiased datatsets generated from random context-free grammar rules. The types of datasets used in our experiments can be divided into two types of random rulesets from which they are generated: a ruleset creating sequences in the form of a binary tree and random context-free grammar rulesets.

The binary-tree context-free grammars are acyclic, such that the maximal sequence length is upper-bounded and each non-terminal symbol only maps to one pair of non-terminals or to a single terminal. The random grammar is generated by simply generating rules at random, and later eliminating all rules that are unproductive or unreachable from the start symbol. The start symbol is also chosen at random.

To avoid problems with memory and runtime, we limit ourselves to a ruleset and sequence length. (more definite numbers). For both grammar types, we consider both small and large versions of datasets. The data associated with an experiment can be divided into three groups, the first being the training data, the second being identically distributed (id), concerning training samples, test data, and the last group being out of distribution (OOD) test data. In this paper, we consider character sequences, whose length conforms to the bounds set during training, to be identically distributed, while either shorter or longer sequences (depending on the experiment) to be OOD.

In all of our experiments, we generate positive samples by recursively parsing each non-terminal symbol according to a randomly chosen production rule, given by the grammar, beginning from the start symbol, until a sequence of terminals is reached. In order to prevent possible divergence of parse trees, we steer the randomness towards terminal symbols, terminating as soon as possible. On the other hand, we sample negative samples by generating a random sequence of terminal symbols, given by the alphabet pertaining to a dataset, and consequently passing it as an input to the classical CYK parser 1. If the parser confirms that the random input sequence is not in the language, the sequence is included as a negative data point. Otherwise, a new sequence is generated and the process is repeated. Finally, in order to obtain meaningful results, we ensure that the test data (both ID and OOD) are only generated via production rules used during the sampling of positive training samples, as the model would otherwise have no way of recognizing positive samples constructed via unseen production rules. Towards this end, the test samples moreover only consist of terminal symbols included in the training set. In table 5 all datasets are described in more detail.

3. Results

Our model is tested using the datasets described above. The numbers of epochs are chosen according to the observed results while running the models and the runtime of the training. We observed the NCYK models to output good results after only about 2 to 10 epochs.

The tables 2, 3, 4 show the accuracy of the different models on all datasets. Hereby, the best accuracy of the last three epochs is chosen. As all datasets are balanced, i.e. contain the same number of positive and negative samples an accuracy of 0.5 indicates an essentially random prediction. We managed to outperform LSTM and Transformer on most of the datasets.

As dimensionality of the NCYK embeddings we choose the number of rules in the grammar from which the dataset was generated, except for the binary datasets. In the latter case, we use a conservative estimate of 125, in order to keep training times reasonable.

Some Transformer results exhibit OOD performance compared to ID. We don't know why this occurs, but assume that this phenomenon may be attributed to the relatively small sizes of our test sets or the strong capabilities of Transformers in handling long sequences.

3.1. Influence of rule size

We would like to show that the "rule size", i.e. the dimensionality of the embedding and intermediate vectors in our model essentially is a general hyperparameter, and

the model still works well, even if the size is chosen very differently than the actual number of rules in a grammar. To that end we show the results of NCYK on the small dataset generated using a random grammar with 20 rules, where we choose the size as 7, 20 and 40 respectively. See table 1 for the obtained accuracies.

#Rules	Distribution	Accuracy
7	ID	0.75
	OOD	0.80
20	ID	0.82
	OOD	0.80
40	ID	0.80
	OOD	0.82

Figure 1. NCYK with different rule number parameter. The actual rule count is 20.

Grammar	Dataset	NCYK	LSTM	TF
Random	ID	**0.82**	0.65	0.72
20 Rules	OOD	0.80	0.55	**0.92**
Random	ID	**0.74**	0.56	**0.74**
31 Rules	OOD	0.71	0.56	**0.83**
Random	ID	0.87	**0.90**	0.85
6 Rules	OOD	**0.90**	0.80	0.78
Binary	ID	**0.94**	0.82	0.76
255 Rules	OOD	**0.90**	0.76	0.58

Figure 2. Results on small datasets

Grammar	Dataset	NCYK	LSTM	TF
Random	ID	**0.98**	0.78	0.83
20 Rules	OOD	**0.94**	0.63	0.88
Random	ID	**0.98**	0.59	0.88
31 Rules	OOD	**0.95**	0.53	0.87
Random	ID	**0.84**	0.83	0.83
6 Rules	OOD	**0.87**	0.85	0.50
Binary	ID	0.93	0.76	**0.99**
2047 Rules	OOD	**0.78**	0.68	0.53

Figure 3. Results on large datasets

4. Discussion

First and foremost, the results show that the proposed NCYK model outperforms the standard neural networks in almost all datasets.

In the small datasets the results are similar to the other two models. For the larger datasets, especially for the inverted one, the standard neural networks start failing on the OOD dataset while our model maintains high accuracy. This outcome underlines the assumption for our model, which is learning the rules of context-free-grammar. Furthermore, the performance of the NCYK model on the inverted

Grammar	Dataset	NCYK	LSTM	TF
Random	ID	**1.00**	0.96	0.94
20 Rules	OOD	**0.85**	0.50	0.50
Random	ID	**0.97**	0.78	0.93
31 Rules	OOD	**0.92**	0.50	0.51
Random	ID	0.93	**0.96**	0.90
6 Rules	OOD	**0.75**	0.55	0.50
Binary	ID	0.93	**0.98**	0.70
255 Rules	OOD	**0.87**	0.76	0.61

Figure 4. Results on inverted datasets

datasets showcases its flexibility, having distinguished short sequences much more accurately than the standard architectures. It is also important to note, that all this is achieved within fewer epochs than the LSTM or Transformer.

We remark that while our model converges far faster, it also is by far slower than the aforementioned methods. This is partially by design, as our method runs in $O(n^3)$, but can also be explained by our implementation being very unoptimized, running mostly directly on Python without using batch processing. More research would be needed to investigate whether NCYK can be made faster and whether it generalizes to non-context-free domains. To that end, we note that the model could potentially also be used for sequence-to-sequence tasks, for example by running it over fixed-length substrings of a sequence or accessing the intermediate hidden vectors.

5. Summary

We have showcased how the classical CYK algorithm can be turned into a trainable deep network, and that this network is then indeed able to perform very well in the original domain of the CYK parser. Even though we achieve better accuracy than Transformer and LSTM based models we consider this for now mainly a theoretically interesting result. While our model reaches very good results on context-free languages, and might potentially generalize to non-context-free domains, its high runtime and memory footprint render it, at least at the moment, rather impractical for longer sequences and large datasets.

References

Chomsky, N. On certain formal properties of grammars. *Information and Control*, 2 (2):137–167, 1959. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(59)90362-6. URL https://www.sciencedirect.com/science/article/pii/S0019995859903626.

Chua, F. C. and Duffy, N. P. Deepcpcfg: Deep learning

and context free grammars for end-to-end information extraction, 2021.

D'Antoni, L. Cs536-s21 intro to programming languages and compilers. URL https://web.archive.org/web/20210719220611/http://pages.cs.wisc.edu/~loris/cs536/slides/lec9.pdf.

Graham, S. L. and Harrison, M. A. Parsing of general context-free languages. volume 14 of *Advances in Computers*, pp. 77–185. Elsevier, 1976. doi: https://doi.org/10.1016/S0065-2458(08)60451-9. URL https://www.sciencedirect.com/science/article/pii/S0065245808604519.

Schlag, I. and Schmidhuber, J. Augmenting classic algorithms with neural components for strong generalisation on ambiguous and high-dimensional data. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*, 2021. URL https://openreview.net/forum?id=_Y4FQu1aJ1Z.

Shi, H., Gao, S., Tian, Y., Chen, X., and Zhao, J. Learning bounded context-free-grammar via lstm and the transformer:difference and explanations, 2022.

Zanzotto, F. M., Satta, G., and Cristini, G. Cyk parsing over distributed representations. *Algorithms*, 13(10):262, October 2020. ISSN 1999-4893. doi: https://doi.org/10.48550/arXiv.1705.08843. URL https://doi.org/10.48550/arXiv.1705.08843.

A. Appendix

A.1. Standard Neural Network Configurations

A.1.1. LSTM AND TRANSFORMER

The input size sequence is the padded sequence and for the last layer to classify the sequences we put a fully connected linear layer.

Hyperparameters LSTM	
num.layers	2
batchsize	25
hidden size	512
learning rate	0.002
optimizer	Adam

Hyperparameters Transformer	
num.layers	2
num.heads	2
batchsize	25
dropout	0.1
hidden size	32
dim feedforward	64
learning rate	0.001
optimizer	Adam

A.2. Datasets

Dataset	Rules	ID length	OOD length	#train	#test id	#test ood
Random-Small 20	20	1-15	30-40	200	100	100
Random-Large 20	20	1-15	30-40	1000	100	100
Random-Inverted 20	20	30-40	1-15	1000	100	100
Random-Small 31	31	1-15	30-40	200	100	100
Random-Large 31	31	1-15	30-40	1000	100	100
Random-Inverted 31	31	30-40	1-15	1000	100	100
Random-Small 6	6	1-15	30-40	200	100	100
Random-Large 6	6	1-15	30-40	1000	100	100
Random-Inverted 6	6	30-40	1-15	1000	100	100
Binary-Small	255	1-20	40-100	200	50	50
Binary-Inverted	255	30-50	1-20	200	50	200
Binary-Large	2047	1-50	100-300	2000	250	250

Figure 5. Dataset Information