Recurrent Neural Networks Meet Context-Free Grammar: Two Birds with One Stone

1st Hui Guan
Computer Science
University of Massachusetts, Amherst
Amherst, MA, USA
huiguan@cs.umass.edu

2nd Umang Chaudhary
Computer Science
University of Massachusetts, Amherst
Amherst, MA, USA
uchaudhary@umass.edu

3rd Yuanchao Xu
Computer Science
North Carolina State University
Raleigh, NC, USA
yxu47@ncsu.edu

4th Lin Ning
Computer Science
North Carolina State University
Raleigh, NC, USA
lning@ncsu.edu

5th Lijun Zhang
Computer Science
University of Massachusetts, Amherst
Amherst, MA, USA
lijunzhang@cs.umass.edu

6th Xipeng Shen
Computer Science
North Carolina State University
Raleigh, NC, USA
xshen5@ncsu.edu

***Abstract*—Recurrent Neural Networks (RNN) are widely used for various prediction tasks on sequences such as text, speed signals, program traces, and system logs. Due to RNNs' inherently sequential behavior, one key challenge for the effective adoption of RNNs is to reduce the time spent on RNN inference and to increase the scope of a prediction. This work introduces *CFG-guided compressed learning*, an approach that creatively integrates Context-Free Grammar (CFG) and online tokenization into RNN learning and inference for streaming inputs. Through a hierarchical compression algorithm, it compresses an input sequence to a CFG and makes predictions based on the compressed sequence. Its algorithm design employs a set of techniques to overcome the issues from the myopic nature of online tokenization, the tension between inference accuracy and compression rate, and other complexities. Experiments on 16 real-world sequences of various types validate that the proposed *compressed learning* can successfully recognize and leverage repetitive patterns in input sequences, and effectively translate them into dramatic (1-1762×) inference speedups as well as much (1-7830×) expanded prediction scope, while keeping the inference accuracy satisfactory.**

***Index Terms*—recurrent neural networks, data compression, context free grammar, tokenization**

I. Introduction

Recurrent Neural Network (RNN) is very effective in modeling and predicting temporal sequences. It has been successfully applied to a broad range of machine learning tasks. Because of RNN's high prediction accuracy, there is also an increasing interest in applying RNNs for sequence prediction tasks in other domains such as program analysis [1], data prefetching and cache placement in computer architecture [2], [3], memory management [4], network caching policy design [5], and system log analysis [6]. As tasks in these domains have real-time or near real-time requirements, speeding up RNN inference is an important problem.

Due to RNNs' inherently sequential behavior, reducing the time spent on RNN inference is a challenging problem. Although many efforts have been taken to accelerate RNN inference, for example by designing efficient model architectures [7], model compression [8], and many other approximations [9], the demands for higher speed remain as the application domains and data volume for RNN keep expanding dramatically. Our study showed that a 1-layer RNN model takes milliseconds to predict the next event on GPUs while prediction tasks in computer systems such as data prefetching and cache replacement typically need results in nanoseconds. The performance issue becomes worse when larger models are used to achieve higher accuracy.

Moreover, demands for long-term large-scope predictions are increasingly popular for RNN. Rather than predicting only the next event, many uses of RNN desire predictions of the next N ($N > 1$) events so that they can start the preparations or take actions earlier. That is especially important if the response (e.g., prefetching or system migration) takes time. There are some attempts to enable large-scope predictions [10]–[12], but they are mostly from the traditional angle, trying to adjust the RNN model architecture or hyperparameters. The prior efforts in pursuing the two important objectives of RNN inferences, improving its speed and scope, have been largely going separately. Large room for improvement remains in both.

In this paper, we present *CFG-guided compressed learning*, a novel method that, by integrating CFG and online tokenization into RNN inference, simultaneously improves the state-of-the-art on both objectives significantly. Unlike popular Deep Neural Network (DNN) compression which compresses *DNN models*, CFG-guided compressed learning compresses *input data sequences*. It is applicable to sequences that consist of many repeated subsequences. For instance, data from sensors in a factory may show similar patterns along time; system logs can have the same event sequences due to repeated operations; the execution traces of a program often manifest repeated patterns. The *basic rationale* is to compress the data sequence by automatically identifying and reducing

the repeated subsequences to an abstract format (i.e., a non-terminal symbol in CFG). If the learner can directly learn and make predictions on the compressed sequence, it may benefit from the identified repetitions in both inference speed and prediction scope.

There are three research questions (RQ) for realizing the idea effectively:

- **RQ1**: How to compress a sequence to keep its statistical properties such that RNNs can still learn patterns from the compressed sequence?
- **RQ2**: How to conduct inference on an online generated data sequence (that is not compressed) given that the model is trained on the compressed sequences?
- **RQ3**: How to support continual model refinement in an online fashion?

This paper presents the first known solution to these open questions by proposing *CFG-guided compressed learning*. It uses no domain knowledge and hence stays completely domain-independent. It learns from compressed sequences and predicts, at one time, not one single event but a sequence of events, achieving both large speedups and also large prediction scopes. It, meanwhile, offers an easy-to-use knob allowing users to keep model accuracy at a satisfying level while enjoying the speed and scope benefits.

CFG-guided compressed learning achieves these by introducing CFG and online tokenization into RNN inference. Specifically, it answers **RQ1** by employing CFG to compactly represent the input data sequence while keeping it in a form amenable for RNN-based learning. It does it by building on an existing linear-time hierarchical compression algorithm, Sequitur [13]. Both RNN training and inference can operate on the CFG representation smoothly. It answers **RQ2** by enabling on-the-fly *incremental compression* via online tokenization as new events arrive and, if necessary, calls the RNN model to make predictions based on the tokenized event sequence. Each prediction is a token in the dictionary, which can be a terminal (one single upcoming event) or a non-terminal (a sequence of upcoming events). It answers **RQ3** through *continuous compression-based refinement* which refines the RNN model on the compressed sequence continuously and efficiently.

Overall, this work makes the following main contributions:

- To the best of our knowledge, this is the first work integrating CFG and sequence compression into RNN for both faster prediction and larger prediction scope on streaming inputs.
- It proposes CFG-guided compressed learning as a novel learning paradigm for RNN-based sequence modeling. The algorithm is applicable to domains whose data sequences have repetitive patterns.
- It overcomes the issues caused by the myopic nature of online tokenization through *efficient rollback*, addresses the tension between compression rate and inference accuracy through *accuracy-conscious lowering*, and minimizes runtime overhead through *partial compression*.
- It empirically validates the benefits of *compressed learning* in improving both the prediction scope and the inference speed of RNN.

II. Compressed Learning Algorithm

Compressed learning learns from compressed sequences either offline or online, and predicts not one single event but a sequence of events. It builds on the grammar-based compression algorithm Sequitur, which (incrementally) compresses a sequence into a Context-Free Grammar (CFG). The learning and inference operate on a variant of the CFG. Compressed learning has three stages: (1) offline compression of the training sequences to build the vocabulary and compressed sequences, (2) offline RNN training using compressed sequences, and (3) online RNN inference and optional online model refinements. The three stages are compatible with the typical workflow for RNN-based application development. In this section, we first introduce major complexities for algorithm design, and then explain the general algorithm.

A. Issues for Algorithm Design

To make the compressed learning algorithm work in general cases, we must address several issues.

Issue-1: Myopic nature of online tokenization. Tokenization is short-sighted. Consider a simple example that has a dictionary with only two entries:

T1: ab T2: abc

For an input sequence `abcabc`, suppose that the RNN predicts `T1` at the starting point. As the tokenizer sees the first two events `ab`, it tokenizes them into token `T1`, and feeds it to the RNN. The RNN would then update its hidden state and make a prediction of the next token, say another `T1`. But when the third event `c` arrives, the tokenizer may realize that the first two events `ab` are actually part of a larger token `T2` (for `abc`). So for the RNN to make predictions based on `T2`, the compressed learning must be able to deal with the premature tokenizations and allow the RNN to undo its state changes when necessary. Such an issue may appear whenever some tokens in a directory are the prefixes of other tokens.

Issue-2: Runtime compression overhead. To refine the RNN at runtime, online compression is needed to generate the compressed sequences so that they can be used to continuously train the RNN model on the fly. Although the refinement is only optional, it is still necessary to minimize the runtime overhead in online compression to maximize the performance benefits of compression learning.

Issue-3: Traps of large tokens. Although a large token could help enable large-scope predictions for its representation of a long subsequence, it could also form a trap. It is because large tokens tend to appear less frequently in the compressed sequence, which makes it harder for RNN to learn about the patterns in the compressed sequence. The compressed learning hence must be able to deal with the tradeoff between token granularities and frequency.

B. Algorithm

In this part, we present the full algorithm of compression learning while highlighting how the three main issues are addressed in the design.

The first two stages in compressed learning produce a vocabulary V and an offline-trained RNN model M. The vocabulary V contains both non-terminal symbols V^N in compressed sequences and all the terminal symbols (i.e., unique events) V^E in uncompressed sequences, i.e., $V = V^N \cup V^E$. Each non-terminal symbol represents a subsequence of events, $v^N = v_1^E, \cdots, v_{|v^N|}^E$, where $v_i^E \in V^E$. For the description purpose, we simplify the notation of an event v^E to e, and refer to both $v^N \in V^N$ and $e \in V^E$ as a *token*. As the first two stages are straightforward applications of the Sequitur compression and standard RNN training, we focus our discussion on the third stage.

Problem definition. The problem of the online prediction is that, given an already emitted sequence of events $s = e^1, \cdots, e^t$, our trained model M shall be able to predict the upcoming events $v_{t+1}^* = e^{t+1}, \cdots, e^{t+|v_{t+1}^*|} \in V$ such that:

$$v_{t+1}^* = \arg\max_v Pr(e^1, \cdots, e^t, v|M), \qquad \text{(II.1)}$$

where $Pr(.|M)$ calculates the probability of the occurrence of a sequence given model M. Here, v_{t+1}^* can be either a non-terminal symbol that represents a sequence of events or a terminal symbol that represents a single event.

Algorithm description. Figure 1 outlines the online inference and model refinement algorithm of compressed learning. Specifically, at a newly emitted event e, the following happens.

1) Tokenization: The algorithm (line 24 in Figure 1) tokenizes e in the context of the earlier events. For a given sequence, a finite state machine (F in Figure 1 line 15) tries to find a token in the vocabulary, the content of which matches with the given sequence. The tokenization subroutine appends the recognized token to the end of the tokenized sequence C; sometimes its old suffix may need to be replaced because a longer match is found.

Rollback (solution to *Issue-1*). The tokenizer helps track the starting point (C.cursor) of the part of C that has not yet been fed into the predictive model M. The replacement of C's suffix (the part following ♦) in tokenization could necessitate the update of the cursor. If the current position of the cursor is in the suffix replaced by the new token, the cursor is updated to the position right before the new token. To make M be able to overcome the premature tokenization and conduct predictions based on the new token, compressed learning records the recent hidden states of M in memory so that M can easily *rollback* its hidden state to the state it had at the new cursor position.

2) Prediction when necessary: After getting the new token, the algorithm (line 26 in Figure 1) checks whether it is time to make a prediction. There are two cases when a new prediction happens: (a) the predicted event for this time point does not match the newly arrived event, which indicates a prediction error; (b) the predicted sequence ends at this time point. In other cases where the prediction is correct so far and the next event is already covered by the recent prediction, there is no need to make a new prediction.

```
1.  // Predict and learn with compression
2.  Input:
3.    P: trace generator
4.    V: initial vocabulary
5.    M: initial predictive model
6.  Output:
7.    M: updated predictive model
8.    V: updated vocabulary
9.  Constants:
10.   START, EOF: markers of the start and end of input
11.   L: length of a learning interval
12.   FREQ: the minimum frequency for a word to get into the vocabulary
13.
14. // create a tokenizer F to recognize the token in V
15. F = tokenizerCreation (V)
16. n = 0  // count the number of events
17. C = emptyList  // store the tokenized sequence
18. i = 0
19. v = M.predict( START )  // predict the upcoming subsequence of events
20. C.cursor = 1 // track the end of the part of C that has been used by M
21. while (e = P.generate () != EOF) { // a new event is produced
22.     n ++
23.     // recognize the new token and update C, M
24.     Tokenize (F, e, C, M)
25.   // if e doesn't match the predicted or the prediction is exhausted
26.     if (!matches(e, v[i]) || i == v.len-1 ) {
27.       // predict the next token (i.e., a subsequence)
28.       v = M.predict ( C[C.cursor : C.len] )
29.       C.cursor = C.len
30.       i=0
31.     }
32.     else { // no prediction needed
33.       i ++
34.     }
35.     if (n == L){
36.       // compress the tokenized sequence seen so far,
37.       // update tokenzied sequence and return new tokens V_
38.       V_ = PartialCompress(C)
39.       M.train (C) // update the predictive model M
40.       F.update (V_) // update the tokenizer with the new words
41.       V.append (V_)  // update the vocabulary
42.       C = []
43.       n = 0
44.   }
45. }
```

Fig. 1. Algorithm of compressed learning for online inference and optional model refinement.

3) Model refinement: Compressed learning supports continuous model refinement. After a certain interval, the algorithm refines the predictive model with the compressed sequence of that interval, as lines 35 to 44 in Figure 1 shows.

Partial compression (solution to Issue-2). The learning starts with compressing the new subsequences in C. A basic design is to run Sequitur on the entire sequence C. But as the tokenizations already compress some parts of the sequence, subroutine *PartialCompress* (line 38) compresses only the uncompressed parts which could save compression time. The subroutine first extracts out all the new subsequences in C that do not match non-terminal tokens. Rather than running Sequitur on each of them, our design is to concatenate them together such that one run of Sequitur would suffice. It is important to notice that simple concatenation can cause wrong compression results, as the subsequences are not actually consecutive but Sequitur could be misled by the concatenated sequence to group the end of a subsequence and the start of another subsequence into one token. To avoid the issue, we insert distinctive symbols at the end of a subsequence as

TABLE I
SEQUENCE STATISTICS. (EVERY SEQUENCE CONTAINS 500K EVENTS.)

Sequences		Compression	#non-terminal	token length stats		
No.	Name	ratio (X)	symbols	min	mean	max
1	fluid-calls	3759.4	6	4	1507.3	8192
2	go-calls	12.2	436	2	14.3	80
3	molecule-calls	96.0	155	2	78.2	1024
4	perl-calls	79.8	116	2	88.4	1880
5	ocean-calls	747.4	27	2	293.7	2194
6	waves-calls	2066.1	16	2	1051.1	8192
7	fluid-mem	2487.6	8	2	1128.9	5120
8	go-mem	86.2	30	2	339.8	3216
9	molecule-mem	4.3	980	2	10.6	85
10	ocean-mem	5.0	916	2	11.7	71
11	perl-mem	13.4	216	2	36.0	577
12	waves-mem	3.5	29	2	16.4	88
13	windows-log1	28.7	213	2	53.3	914
14	windows-log2	29.7	269	2	34.3	469
15	thunderbird-log1	17.0	403	2	22.1	2048
16	thunderbird-log2	20.9	428	2	17.1	1536

* The frequency threshold in the lowering step is set to 5 when the reported statistics are collected. *token length stats* consider only non-terminals in the compressed seq; a token is a sequence of events. *X-calls*: function call seq.; *X-mem*: memory address traces; *X-log*: system logs.

separators,

Accuracy-Conscious Lowering (solution to Issue-3). Lowering is an important step for striking a good tradeoff between token granularity and frequency. It, from the CFG, derives a compressed sequence friendly to RNN training (both offline and online). It recursively conducts a depth-first expansion of tokens in an input compressed sequence (s). If a token's frequency is no smaller than a threshold ($FREQ$), the subroutine stops expanding it, and puts it into the vocabulary as a valid token. Such a design avoids unnecessary expansions to keep the sequence as compact as possible while meeting the frequency requirement. The frequency threshold ($FREQ$) is a hyperparameter that adjusts the tradeoff between the compression rate and the frequency of tokens. During offline training, compressed learning uses binary search to automatically find the suitable frequency threshold that meets a user-specified accuracy requirement.

III. EVALUATION

We conducted a set of experiments to examine the efficacy of the proposed technique, trying to answer the following questions: (1) How much benefit can we get from compressed learning for inference speed and prediction scope? (2) How does compressed learning affect the model quality? (3) What is the runtime overhead of incremental tokenization for online inference?

A. Methodology

Datasets. When collecting traces for the experiments, in order to get a comprehensive assessment of the technique, we try to ensure that the traces (i) come from the real-world workloads or systems; (ii) exhibit a spectrum of regularities; (iii) cover several different types of events and domains.

Table I lists the sixteen traces we experiment with. They are of three types: The first six are function call sequences, the second six are memory address traces (in 64-byte data blocks), and the final four are system log traces. Prediction on these sequences can help guide just-in-time optimizations, prefetching, and system anomaly detection.

Counterparts for comparisons. Since CFG-guided compressed learning is generally applicable to domains whose sequences have repetitive patterns, we use standard RNN-based sequence modeling used in these domains as our baselines. Specifically, we compare our compressed learning (denoted as *ours*) with the following two default approaches.

(1) *Default learning with 1-event prediction (default-1).* This method trains the RNN using the *un-compressed sequence* and predicts only the next single event at one prediction. The number of predictions it has to make is the same as the number of events in a test sequence.

(2) *Default learning with k-event prediction (default-k).* This method trains the RNN using the *un-compressed sequence* but has the same prediction scope as our *compressed learning* has.

Models. The RNN model used in the experiments of all the methods is the same. It consists of an embedding layer with an embedding dimension of 256, a GRU layer with 1024 units, and a fully-connected output layer. We train an RNN model for each sequence. For offline training, the RNN models are trained with ADAM using an input length of 100 for all methods. If online training is enabled, the models are refined for one epoch on each interval (i.e., 50,000-length event sequence) with an input length of 100.

Hyperparameters. Compared to default RNN training, the only extra hyperparameter introduced by compressed learning is the frequency threshold ($FREQ$) used in the lowering step. We used binary search to determine the best $FREQ$ that meets a user-specified accuracy requirement while achieving good inference speedups.

Metrics. Our evaluation uses the following three metrics. (i) The *speedup* over the inference time (i.e., averaged time spent on predicting the next event) taken by *default-1* when all runtime overhead is counted in. (ii) The *prediction scope*, which is the average length of a prediction. (iii) The *prediction accuracy*, which is the ratio between the # of correctly predicted events over the total number of events.

B. Results

Table II reports the online prediction results of compressed learning and its comparison with the two default approaches. The user-specified tolerable accuracy drops are 0% and 1%, with respect to *default-1*. The results are averaged over five runs with different random seeds. Standard deviation of event accuracy varies from zero to 1.9%.

Table II shows the clear benefits from our compressed learning on both prediction scope and speed. The prediction scope increases from one in *default-1* to hundreds or even thousands of events (as the "avg. pred length" column shows), and the inference time decreases by up to three orders of magnitude

TABLE II
ONLINE PREDICTION RESULTS OF COMPRESSED LEARNING AND ITS COMPARISON WITH DEFAULT APPROACHES THAT USE UNCOMPRESSED SEQUENCES.

Sequences	spec. acc. drop	$FREQ$	avg. pred length	#predictions	#rollbacks	tokenization overhead (%)	avg. latency* (ms)		prediction speedup (×)	event accuracy (%)		
							ours	default-1		ours	default-k	default-1
fluid-calls	0%	2	7830	35	0	0.20680	0.036	4.340	120.9	99.9984	99.9984	99.9984
	1%	2	7830	35	0	0.20680	0.036		120.9	99.9984	99.9984	
perl-calls	0%	50	31	8916	522	0.00017	0.132	3.701	28.1	99.74	98.54	99.89
	1%	5	133	7372	1180	0.00018	0.082		45.3	99.57	98.59	
molecule-calls	0%	50	30	8572	722	0.00044	0.179	3.582	20.0	99.67	95.86	99.6
	1%	5	91	4393	1099	0.00287	0.071		50.5	99.5	77.37	
ocean-calls	0%	500	17	14677	191	0.00007	0.209	3.726	17.9	99.89	99.51	99.95
	1%	5	200	5136	725	0.00282	0.084		44.5	98.94	44.59	
wave-calls	0%	20	4798	4070	21	0.00616	0.246	3.707	15.1	83.74	83.76	83.76
	1%	20	4798	4070	21	0.00616	0.246		15.1	83.74	83.76	
go-calls	0%	20000	1	248732	2132	0.00001	3.608	3.608	1	87.4	87.59	87.59
	1%	5000	2	229512	3767	0.00001	3.280		1.1	86.68	70.47	
fluid-mem	0%	5	2500	100	0	0.05676	0.002	3.551	1762	99.96	89.53	99.97
	1%	5	2500	100	0	0.05676	0.002		1762	99.96	89.53	
go-mem	0%	5	78	3336	55	0.00001	0.054	3.611	66.4	98.82	91.49	99.04
	1%	5	78	3336	55	0.00002	0.054		66.4	98.82	91.49	
perl-mem	0%	20	81	6273	105	0.00014	0.086	3.630	42.2	99.48	98.52	98.74
	1%	20	81	6273	105	0.00014	0.086		42.2	99.48	98.52	
ocean-mem	0%	20	4	83383	3640	0.00002	1.478	3.954	2.7	81.63	75.93	81.3
	1%	5	6	78782	1166	0.00004	1.236		3.2	80.69	72.37	
wave-mem	0%	500	2	193050	1902	0.00001	3.084	3.700	1.2	79.32	64.71	79.56
	1%	5	5	85957	1416	0.00001	1.233		3	79.1	50.71	
molecule-mem	0%	2000	1	250000	0	0.00001	3.625	3.625	1	93.3	93.36	93.36
	1%	1000	2	210869	1495	0.00000	3.296		1.1	92.05	73.69	
windows-log1	0%	200	11	40930	3768	0.00004	0.605	3.750	6.2	95.13	88.91	95.91
	1%	20	31	24391	6996	0.00017	0.364		10.3	93.25	79.54	
windows-log2	0%	100	14	38781	4636	0.00005	0.716	4.009	5.6	95.37	90.62	96.49
	1%	20	23	29882	9836	0.00012	0.617		6.5	95.08	91.81	
thunderbird-log1	0%	1000	3	109977	4580	0.00011	1.521	4.014	2.6	93.91	94.02	94.02
	1%	200	10	40933	2445	0.00026	0.628		6.4	92.37	84.93	
thunderbird-log2	0%	20000	1	250000	0	0.00001	3.531	3.531	1	92.45	92.52	92.52
	1%	1000	2	135229	11498	0.00007	1.962		1.8	91.04	83.53	

*avg. latency: averaged time spent on predicting the next event. *default-1* and *default-k* have the same avg. latency.

(as the "prediction speedup" column shows). Getting benefits on both aspects at the same time shall be no surprise. The larger prediction scopes entail the need for fewer predictions, and hence the much-reduced prediction time.

In comparison, when the default method extends its prediction scope to the same as the compressed learning has, significant accuracy loss appears (e.g., 54% accuracy loss on `ocean-calls`), as the "default-k" column shows. Moreover, to predict k events, *default-k* still needs to make k predictions; so it saves no prediction time at all.

The exact amount of speedups by compressed learning varies from sequence to sequence, depending on how often repetitive patterns show up in the sequence, which is intuitive. What is satisfying is that for traces with regular patterns, compressed learning can indeed tap into the potential, effectively recognizing the patterns and translating them into dramatic speedups, as typified by the results on the traces of `fluid`. On the other hand, on irregular traces, the method can still achieve the target accuracy while causing no slowdowns, as shown by the function call sequence of `go`, the random tree search application.

The effectiveness of the technique holds across domains and sequence types. The benefits are more pronounced on function call and memory traces than on system logs, due to the less regularity in the system logs. But it is worth noting that even on system logs, the benefits are still significant, 1–10.3× speedups of inference and up to 31× larger prediction scopes. To achieve the same prediction scopes, *default-k* suffers up to 16% accuracy drops while giving no speedups.

Runtime overhead of online tokenization. The myopic nature of online tokenization incurs a number of rollbacks in compressed learning for most sequences, as the "#rollbacks" column in Table II shows. But in fact, rollbacks do not cause extra invocations of predictions. The time overhead of a rollback consists of only the switch of one single reference (to point to an earlier data block that holds the recent hidden state of the RNN), which is negligible. That explains the significant speedups despite the many rollbacks in compressed learning.

The other source of runtime overhead is the time spent on tokenization for online prediction. The results are listed in column "tokenization overhead (%)" of Table II. Overall, this tokenization overhead is negligible, less than 0.2% compared to the total amount of prediction time (the time spent on the RNN model plus online tokenization) for all sequences.

IV. RELATED WORK

Deep learning on compressed inputs. There are some studies on deep neural network (DNN) training and inference with compressed input data, but all on images and convolutional

neural network (CNN) [14]–[16]. In Natural Language Processing (NLP), the representation of inputs sometimes uses some tokens to represent some common phrases. An example is Byte Pair Encoding (BPE) [17] used in subword tokenization. These representations are at the word or phrase level, offering no systematic ways to identify patterns in a long sequence of events and code them concisely. Moreover, as those studies work on separate sentences instead of continuous event streams, rather than online tokenizing inputs continuously, they use a preprocessing step to first tokenize the entire sentence before feeding it to the DNN. They are not applicable to streaming event sequences. To the best of our knowledge, this work gives the first proposal of compressed learning for RNNs on streaming event sequences.

DNNs for program traces. Some recent works have proposed applying DNNs on program traces for program behavior prediction. A study [3] uses an offline attention-based LSTM model to provide insights for designing a simple online hardware cache replacement policy. Another study [2] applies sequence learning to prefetching and proposes using LSTM to understand the semantic information of the underlying application given a memory access trace. A recent work [4] proposes an RNN-based page scheduler for programs that execute over hybrid memory systems. None of them have considered learning from the compressed traces.

DNNs for system logs. Recent years have seen a growing interest in applying Deep Learning models in analyzing system logs. One study [6] proposes Deeplog, which leverages LSTM for online anomaly detection. Another work [18] proposes to use RNN with the attention mechanism for anomaly detection. Some other work [5] builds an RNN-based content caching framework to predict the popularity of content objects on information-content networks. Wang and others [19] used RNNs to predict the probability that a user will access a particular activity given their historical access logs. No prior work has proposed learning from compressed log sequences.

V. Conclusion

This paper presents *CFG-guided compressed learning*, the first known approach to integrating sequence compression into RNN learning and inference for both expanded prediction scope and reduced inference latency. It builds on CFG and online tokenization, and addresses a series of complexities through the design of efficient rollback, accuracy-conscious lowering, partial compression, and other techniques. By discovering and leveraging patterns in a sequence effectively, it enables much faster inferences while achieving a substantially expanded prediction scope on 16 real-world sequences with repetitive patterns. Future work includes generalizing compressed learning to other autoregressive models and recent architectures such as Transformers.

VI. Acknowledgements

This material is based upon work supported by the UMass Startup Fund, the National Science Foundation (NSF) under Grants CNS-1717425, CCF-1703487, CCF-2028850, and the Department of Energy (DOE) under Grant DE-SC0013700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

References

[1] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4505–4515.

[2] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *International Conference on Machine Learning*, 2018.

[3] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[4] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 37–48.

[5] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "Deepcache: A deep learning based framework for content caching," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018, pp. 48–53.

[6] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[7] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased lstm: Accelerating recurrent network training for long or event-based sequences," in *Advances in neural information processing systems*, 2016, pp. 3882–3890.

[8] M. S. Zhang and B. Stadie, "One-shot pruning of recurrent neural networks by jacobian spectrum evaluation," *arXiv preprint arXiv:1912.00120*, 2019.

[9] L. Liu, L. Deng, Z. Chen, Y. Wang, S. Li, J. Zhang, Y. Yang, Z. Gu, Y. Ding, and Y. Xie, "Boosting deep neural network efficiency with dual-module inference," 2020.

[10] S. H. Park, B. Kim, C. M. Kang, C. C. Chung, and J. W. Choi, "Sequence-to-sequence prediction of vehicle trajectory via lstm encoder-decoder architecture," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 1672–1678.

[11] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016.

[12] H. Cheng, P.-N. Tan, J. Gao, and J. Scripps, "Multistep-ahead time series prediction," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2006, pp. 765–774.

[13] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: a linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[14] L. Gueguen, A. Sergeev, B. Kadlec, R. Liu, and J. Yosinski, "Faster neural networks straight from jpeg," in *Advances in Neural Information Processing Systems*, 2018, pp. 3933–3944.

[15] Z. Liu, T. Liu, W. Wen, L. Jiang, J. Xu, Y. Wang, and G. Quan, "Deepn-jpeg: A deep neural network favorable jpeg-based image compression framework," in *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[16] X. Xie and K.-H. Kim, "Source compression with bounded dnn perception loss for iot edge computer vision," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.

[17] P. Gage, "A new algorithm for data compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

[18] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*, 2018, pp. 1–8.

[19] H. Wang, Z. Wang, and Y. Ma, "Predictive precompute with recurrent neural networks," *arXiv preprint arXiv:1912.06779*, 2019.