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

HUI GUAN and UMANG CHAUDHARY, College of Information and Computer Sciences, University of Massachusetts, Amherst, Massachusetts, USA
YUANCHAO XU and LIN NING, North Carolina State University, Raleigh, North Carolina, USA
LIJUN ZHANG, College of Information and Computer Sciences, University of Massachusetts, Amherst, Massachusetts, USA
XIPENG SHEN, North Carolina State University, Raleigh, North Carolina, USA

This work addresses a key challenge in the effective adoption of Recurrent Neural Networks (RNNs) by reducing inference time and expanding the scope of a prediction. It introduces compressed learning, a novel approach that integrates Context-Free Grammar (CFG) and online tokenization into the training and inference of RNNs 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 in sequence compression and prediction. Its effectiveness is theoretically analyzed and empirically validated on 16 real-world sequences, including program function calls, memory traces, and system logs. Empirical results demonstrate that compressed learning can successfully recognize and leverage repetitive patterns in input sequences, and effectively translate them into dramatic (1–1,762×) inference speedups as well as much (1–7,830×) expanded prediction scope, while keeping the inference accuracy satisfactory.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Theory of computation** → **Grammars and context-free languages**;

Additional Key Words and Phrases: recurrent neural networks, data compression, context free grammar, tokenization

Associate Editor: Suhang Wang

ACM Reference format:
Hui Guan, Umang Chaudhary, Yuanchao Xu, Lin Ning, Lijun Zhang, and Xipeng Shen. 2026. Recurrent Neural Networks Meet Context-Free Grammar: Two Birds with One Stone. *ACM Trans. Knowl. Discov. Data.* 20, 2, Article 29 (February 2026), 22 pages.
https://doi.org/10.1145/3785464

Authors' Contact Information: Hui Guan (corresponding author), College of Information and Computer Sciences, University of Massachusetts, Amherst, Massachusetts, USA; e-mail: huiguan@cs.umass.edu; Umang Chaudhary, College of Information and Computer Sciences, University of Massachusetts, Amherst, Massachusetts, USA; e-mail: uchaudhary@umass.edu; Yuanchao Xu, North Carolina State University, Raleigh, North Carolina, USA; e-mail: yxu47@ncsu.edu; Lin Ning, North Carolina State University, Raleigh, North Carolina, USA; e-mail: lning@ncsu.edu; Lijun Zhang, College of Information and Computer Sciences, University of Massachusetts, Amherst, Massachusetts, USA; e-mail: lijunzhang@cs.umass.edu; Xipeng Shen, North Carolina State University, Raleigh, North Carolina, USA; e-mail: xshen5@ncsu.edu.

ACM 1556-472X/2026/2-ART29
https://doi.org/10.1145/3785464

1 Introduction

Recurrent Neural Network (RNN) [23] is very effective in modeling and predicting temporal sequences. It has been successfully applied to a broad range of sequence prediction tasks such as program analysis [38], data prefetching and cache placement in computer architecture [28, 49], memory management [21], network caching policy design [40], system log analysis [22], recommender systems [6], and many others [14, 53]. Although transformer-based architectures [55] have recently been shown to be a powerful alternative, RNNs still shine in scenarios where large-scale data, such as memory traces [32] and system logs [47], are not accessible. As tasks in these domains have real-time or near real-time requirements, speeding up RNN inference is an important problem.

Due to auto-generative behaviors in RNN inference, reducing the inference time is a challenging problem. Although many efforts have been taken to accelerate RNN inference, for example by designing efficient model architectures [41], model compression [62], sparsification [63], and many other approximations [35], the demands for higher speed remain. Our study showed that a one-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.

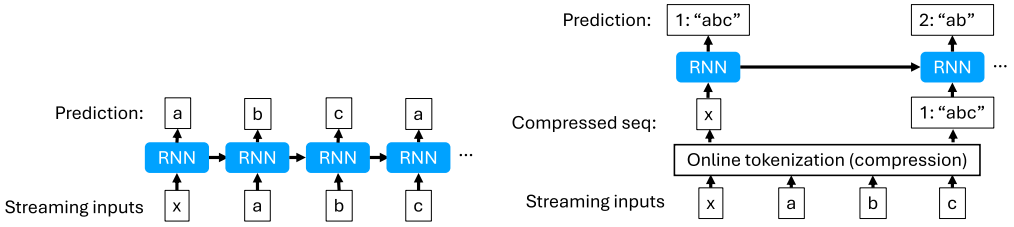
Moreover, demands for long-term large-scope predictions are increasingly popular for RNN. Rather than predicting only the next event, many uses of RNN want to predict the next N ($N > 1$) events so that they can start preparations or take actions earlier. This is especially important if the response (e.g., prefetching or system migration) takes time. There are some attempts to enable large-scope predictions [13, 45, 59], but they are mostly from the traditional angle, trying to adjust the RNN model architecture or hyperparameters. Prior efforts to pursue the two important objectives of RNN inferences, *improving its speed and scope*, have largely been made separately. In both, there is great room for improvement.

To address the problems, this article presents *compressed learning*, a novel method that, by integrating **Context-Free Grammar (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*, 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 the inference speed and the prediction scope.

Figure 1 illustrates the difference between default RNN inference and the proposed RNN inference. In default RNN inference, the RNN processes every individual event in the streaming inputs and predicts the next event. In contrast, compressed learning enables the RNN to process a compressed sequence, generated through online tokenization, which is often significantly shorter than the original sequence. This approach allows the RNN to predict the next subsequence, which may represent a single event or a sequence of events. Compressed learning enhances inference efficiency by reducing the number of predictions required for streaming inputs while simultaneously expanding the prediction scope.

There are three **Research Questions (RQ)** for realizing the idea effectively.

— *RQ1*: How to compress a sequence to keep its statistical properties so that RNNs can still learn patterns from the compressed sequence?



(a) Default RNN inference (b) Proposed RNN inference: compressed learning

Fig. 1. Default RNN inference (a) vs. the proposed RNN inference (b): In (a), the vocabulary consists of {“x,” “a,” “b,” “c”}, where each symbol represents a single event. In (b), the vocabulary is extended to {“x,” “a,” “b,” “c,” 1, 2}, where “1” represents the subsequence “abc” and “2” represents the subsequence “ab.” Both subsequences are supposed to be repeated many times in the streaming inputs and thus are abstracted to single symbols. The figure illustrates two benefits of compressed learning: (i) smaller average prediction speed—the number of predictions is reduced from 4 to 2 for a sequence length of 4, and (ii) larger prediction scope—the prediction scope is expanded from a single event to 2–3 events.

— *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?

When answering the questions, it is important to note *three principles*. (i) *Domain-independent.* The solution should work across domains, which is essential for its general applicability for data sequences with repetitive patterns. For example, it is possible to use knowledge of the program code structure to compress program traces [31], but such a method cannot apply to sensor data, health data, or data in many other domains, and therefore it does not fit the need. (ii) *Beneficial.* The overhead of the solution should not outweigh its benefits. (iii) *Staying accurate.* User satisfaction with model accuracy should not be the price for the improvement of inference speed and scope.

This article presents the first known solution to these open questions by proposing *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 large prediction scopes. It, meanwhile, offers an easy-to-use knob that allows users to keep the model accuracy at a satisfying level while enjoying the speed and scope benefits.

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 [42]. 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 vocabulary, which can be a terminal (one single upcoming event) or a non-terminal (a sequence of upcoming events). It answers *RQ3* through *partial compression*, which leverages results from online tokenization to efficiently generate compressed sequences for online model updates. Section 3 explains the algorithm in detail.

Section 4 reports experiments on 16 real-world data sequences including program function calls, memory traces, and system logs. The results show that compared to RNN predictors, compressed learning achieves 1–1,762× prediction speedups for its large prediction scopes (up to 7,830 events per prediction). For the same prediction scope, compressed learning gives as much as 54% higher

prediction accuracy than default RNN predictions. When online training is enabled, we demonstrate that the compression overhead from partial compression is minimal, accounting for up to 6.08% of the per-epoch model training time.

Overall, this work makes the following main contributions.

- —It proposes compressed learning as a novel learning paradigm for RNN-based sequence modeling with the goal of both faster prediction and larger prediction scope on streaming inputs. 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 analytically studies the efficiency benefits of *compressed learning*, and empirically validates its benefits in improving both the prediction scope and the inference speed of RNN.

This work is an extension of a prior work accepted to the ICDM'21 [26], which is a six-page short paper with references included. It extends the short paper significantly as follows. (1) We add the background on CFG to make the article self-contained (Section 2). (2) We add a running example to illustrate the compressed learning algorithm (Section 3.1). (3) We add theoretical analysis to show the efficiency benefits of compressed learning (Section 3.4). (4) We add experiments to evaluate the benefits of online model refinements and quantify the runtime overhead of partial compression (Section 4.3). (5) We refine the presentation throughput the article (e.g., adding principles we follow to devise the algorithm, discussions on accuracy-conscious lowering, descriptions on datasets, visualizations).

2 Background

CFG is a formal grammar that consists of production rules. Each production rule is of the form $A \rightarrow \beta$, where A is a single *non-terminal* symbol and β is a sequence of terminals and non-terminals. It is context-free because the **Left-Hand Side (LHS)** of the production rule can always be replaced with the **Right-Hand Side (RHS)** regardless of the context of the LHS (non-terminal). To compress a data sequence, one can transform the sequence into a CFG. The process is called *grammar-based compression*.

Sequitur is a classic grammar-based compression algorithm, originally proposed by Nevill-Manning and Witten [42]. It infers a hierarchical structure from a sequence of discrete symbols in linear time. These discrete symbols are treated as terminal symbols. For a given sequence of symbols, it derives a CFG where each rule reduces a repeatedly appearing sequence of terminals into a non-terminal symbol. For example, if the sequence is "abcab," the algorithm produces CFG: `S -> AcA, A -> ab`. By substituting repeating strings (e.g., "ab") with new non-terminal symbols (e.g., "A"), it produces a concise representation of the input sequence (e.g., "AcA").

We have not seen its use in RNN. This work selects Sequitur as the compression algorithm for the fit of the compression results for RNN and its domain-independent property. Some similar grammar-based compression algorithms [8–10] could be used as well.

3 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 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

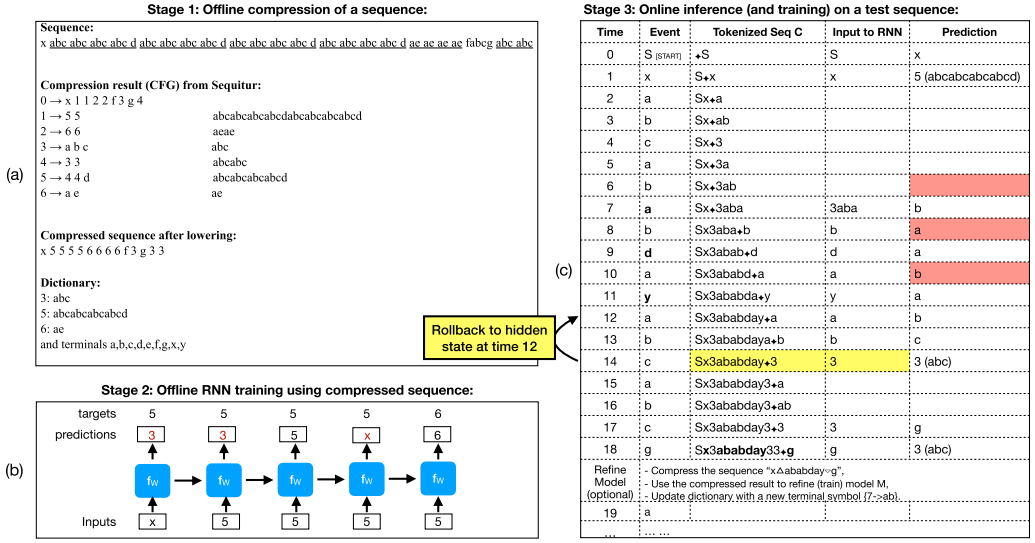


Fig. 2. Running example of compressed learning: (a) offline compression of a train sequence, (b) offline RNN training using compressed sequence with input length of five and batch size of one, and (c) online inference and optional model updates on a test sequence. In (c), the diamond symbol ◆ represents "C.cursor" in Algorithm 1, which indicates the start of the subsequence in C that has not been sent to the predictive model M. Cells in the "Prediction" column are filled in red if the predictions are wrong.

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 a running example to illustrate the three stages, and then explain the major complexities and the general algorithm.

3.1 Running Example

We use the example in Figure 2 to convey the intuitions of compressed learning.

Stage 1: Offline Compression. Figure 2(a) illustrates the first stage—that is, how the grammar-based compression works on a sequence. The *data sequence* is compressed by Sequitur into the *CFG* that consists of seven rules. Each rule shows what sequence (on the RHS of the rule) a *non-terminal symbol* (LHS) represents. Rule 0 represents the entire sequence, and each of the other rules represents a subsequence in the entire sequence. For reference, we expand the RHS of each rule (except Rule 0) and put the results to the right of the CFG rules.

Here, we briefly restate the essential mechanics of the Sequitur-based hierarchical compression used in our framework. Sequitur operates by scanning the input sequence from left to right and constructing grammar rules whenever repeated symbols is detected. Each repeated subsequence is replaced with a non-terminal, and the rule defining that non-terminal is added to the grammar. This process is recursive: if the newly created non-terminal participates in further repetitions, additional rules are formed, producing a compact hierarchical grammar. For example, given the sequence "abcabc..." in Figure 2(a), Sequitur first introduces a rule 3 ⟶ *abc* and then a rule 4 ⟶ 33. In compressed learning, we adapt this process by lowering overly large or infrequent tokens to ensure that the resulting compressed sequences remain suitable for RNN training.

Below the CFG is a lowered representation of the RHS of Rule 0 after some symbols in the RHS are expanded. The expansion process is called *lowering*, which makes the compressed sequence

into a form more friendly for RNN training compared to the original RHS of rule 0. Details will be explained in Section 3.3.

Below the lowered sequence is a *dictionary*, which records all the non-terminal symbols that appear in the lowered sequence, along with what subsequence each of them represents. The dictionary, by default, includes all the terminal symbols in the train sequence. The dictionary is called *vocabulary* in the rest of the article.

Stage 2: Offline RNN Training. Figure 2(b) illustrates offline RNN training in compressed learning. The lowered sequence serves as the input for training the RNN-based predictive model. The training process follows standard RNN training procedures but uses compressed sequences as both inputs and targets.

Stage 3: Online Inference and Optional Model Updates. Figure 2(c) shows the inference steps for a test sequence. After the inference starts, at time 1, the arriving event x triggers a prediction by the trained RNN. The prediction gives out a token 5, which corresponds to a subsequence of events abcabcabcabcd represented by the RHS of rule 5 in Figure 2(a). The events coming in the next 5 timesteps (time 2–6) are all consistent with that prediction, and hence the RNN does not need to be invoked to make predictions at those times. Note that as those events arrive, our compressed learning algorithm automatically *tokenizes* them, as illustrated by the replacement of abc with token 3 at time 4.

Formally, given an event sequence s and a vocabulary V, one can replace subsequences of events in s with the corresponding non-terminal symbols in V. The process is called *tokenization* and the resulting sequence is called *tokenized sequence*. Both the RHS of rule 0 in Figure 2(a) and compressed sequence after lowering are tokenized sequence.

At time 7, the actual event a differs from the prediction (c), which prompts the RNN to discard the rest of its prediction, and make another prediction based on the subsequence that has not yet been fed to the RNN—that is, the subsequence following ◆ in the "Tokenized Seq C" column in Figure 2(c), 3aba in this case. The prediction is a single event b, which matches the actual event at time 8.

At time 8, because no prediction is there for the next timestep, the RNN is triggered to make another prediction. The prediction is a single event a, which does not match the actual event at time 9. Therefore, at time 9, the token d is fed into the RNN as the input and the prediction is a for time 10. This process continues.

At time 14, when the event c arrives, it is recognized as part of the longer token 3, representing abc. The sequence abc is replaced by the token 3, which should be fed into the RNN for the next prediction. The hidden state of the RNN must be rolled back to the state at time 12, before processing the token a. In this case, the RNN hidden state is first rolled back to time 12, and then the token 3 is fed into the RNN for the next prediction. The RNN generates another subsequence (abc, represented by token 3) as its prediction, which is verified as correct since abc matches the future events at times 15–17.

After time 18, suppose one intends to refine the model using the newly collected event sequence. To support online training, the new event sequence must first be compressed. We later show that online training can be seamlessly integrated with minimal compression overhead. Compressed learning starts an online compression on the uncompressed event subsequences in Tokenized Seq C and has the RNN learn from the compressed results to get updated, illustrated by the "Refine Model (optional)" row in Figure 2(c). The process then continues. It is important to note that this online training is entirely optional; it can be disabled without compromising the effectiveness of compressed learning in improving inference efficiency.

3.2 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. Online tokenization is essential in Stage 3 to process any uncompressed test sequence, making it compatible with the RNN trained on compressed sequences. However, tokenization can be 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). Therefore, for the RNN to make predictions based on T2, the compressed learning must be able to deal with premature tokenizations and allow the RNN to undo its state changes when necessary. This issue may appear whenever some tokens in a directory are the prefixes of other tokens (e.g., tokens 3 and 5 in the example in Figure 2(a)). This issue raises further questions on the influence of rollbacks on the performance of compressed learning. Would they incur extra invocations of the RNN? Would they cancel out the time savings from compressed learning?

Issue-2: Runtime Compression Overhead. In scenarios where online learning is preferred as new data accumulates, it is crucial for data compression to be efficient to minimize compression overhead in Stage 3. 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. It is important to note that our focus is on enhancing compression efficiency rather than improving online learning algorithms.

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. In Figure 2(a), for instance, the top-level tokens in the RHS of rule 0 each occur at most twice, but in the lowered sequence in the same figure, some tokens (e.g., 5, 6) appear twice as often. The compressed learning hence must be able to deal with the tradeoff between token granularities and frequency.

3.3 Algorithm

In this part, we present the full algorithm of compression learning while highlighting how the three main issues are addressed in the design. Our explanations use the notations listed in Table 1.

The first stage in compressed learning produces a vocabulary V (illustrated in Figure 2(a) as a dictionary) and the compressed sequence after lowering as the training dataset. The vocabulary V will be used for online tokenization for streaming inputs in Stage 3. It 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*.

The second stage produces an offline-trained RNN model M (illustrated in Figure 2(b)). As the first two stages are straightforward applications of the Sequitur compression and standard RNN training, we focus our discussion on the third stage.

Table 1. Notations

Notation	Description
M	A RNN model.
V^E	A set of terminal symbols (i.e., unique events).
V^N	A set of non-terminal symbols.
V	Vocabulary: It contains both V^E and V^N.
C	A tokenized sequence: Each element is a token $v \in V$.
e^t	An event emitted at time t. $e^t \in V^E$.
v^E	A terminal symbol: $v^E \in V^E$.
v^N	A non-terminal symbol: $v^N \in V^N$.
v^*_{t+1}	Predicted token (a sequence of upcoming events) after t, $v^*_{t+1} \in V$.

Problem Definition. The problem of the online prediction in Stage 3 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:

$$v^*_{t+1} = \arg\max_{v} Pr(e^1, \cdots, e^t, v|M), \tag{3.1}$$

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. Algorithm 1 outlines the online inference process and the optional model refinement supported by compressed learning. Specifically, at a newly emitted event e, the following happens (lines 7–28).

3.3.1 Online Tokenization. Online tokenization identifies a token from the vocabulary that ends with the newly emitted event e. For example, using the vocabulary and test sequence shown in Figure 2, when the newly emitted event e is c at time 4, the tokenizer recognizes that c, together with the two preceding events, matches the token 3 in the vocabulary. Consequently, the subsequence abc is replaced with 3.

The algorithm (line 9 in Algorithm 1) tokenizes e in the context of earlier events recorded in C. The tokenizer internally constructs a finite state machine F based on the vocabulary V (line 1 of Algorithm 1) to recognize tokens. For a given sequence, the finite state machine (F, referenced in Algorithm 1 line 9) attempts to identify a token from the vocabulary whose content matches the sequence. The tokenization subroutine appends the recognized token to the end of the tokenized sequence C; in some cases, the old suffix of C may need to be replaced if a longer match is found.

The tokenization algorithm is presented in Algorithm 2. It appends the recognized token (line 1) to the end of the tokenized sequence C (line 12) if the recognized token is the event itself. In some cases, the old suffix of C may need to be replaced if a longer match is found (line 4), potentially triggering the rollback process described in lines 5–19.

Rollback (Solution to Issue-1). The tokenizer tracks 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 ensure M can handle the premature tokenization and make 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. Supporting rollback

Algorithm 1: Online Inference in Compressed Learning

Require: P: trace generator, V: initial vocabulary, M: initial predictive model, $START$, EOF: markers of the start and end of input, L: length of a learning interval, $FREQ$: minimum frequency for a word to be added to the vocabulary
Ensure: M: updated predictive model, V: updated vocabulary

$F \leftarrow$ tokenizerCreation(V) *{Create a tokenizer F to recognize tokens in V}*
$n \leftarrow 0$ *{Count the number of events}*
$C \leftarrow$ emptyList *{Store the tokenized sequence}*
$i \leftarrow 0$ *{Keep track of the index of the predicted events that need to be matched}*
$v \leftarrow M$.predict($START$) *{Predict the upcoming subsequence of events}*
C.cursor $\leftarrow 1$ *{Track the end of the part of C used by M}*
while $(e = P.\text{generate}()) \neq EOF$ **do**
  $n \leftarrow n + 1$ *{A new event is produced}*
  Tokenize(F, e, C, M) *{Recognize the new token and update C, M}*
  if !matches(e, $v[i]$) **or** $i = v$.len $- 1$ **then**
    {If e doesn't match the predicted or the prediction is exhausted}
    $v \leftarrow M$.predict($C[C$.cursor : C.len$]$) *{Predict the next subsequence}*
    C.cursor $\leftarrow C$.len
    $i \leftarrow 0$
  else
    {Else, no prediction needed}
    $i \leftarrow i + 1$
  end if
  if $n = L$ **then**
    {Optional: if model updates is triggered, compress the tokenized sequence seen so far for model updates}
    $V' \leftarrow$ PartialCompress(C) *{Update tokenized sequence and return new tokens V'}*
    M.train(C) *{Update the predictive model M}*
    F.update(V') *{Update the tokenizer with new words}*
    V.append(V') *{Update the vocabulary}*
    $C \leftarrow$ emptyList
    $n \leftarrow 0$
  end if
end while

Algorithm 2: The Tokenize (F, e, C, M) Subroutine in Algorithm 1. It recognizes the New Token That Has e as the Final Element, Update Tokenized Sequence C with the New Token, and also Perform Rollback if Necessary

Require: F: tokenizer, e: new event, C: tokenized sequence, M: predictive model
Ensure: Updated tokenized sequence C and model M

$newToken \leftarrow F$.recognize(e) *{Recognize the token}*
if $newToken$.len > 1 **then**
  {If the recognized token covers more than a single event, need to replace the suffix of C}
  replaceSuffix(C, $newToken$) *{Replace the ending tokens in C with $newToken$}*
  if C.cursor $> C$.len $- 1$ **then**
    {Rollback is needed}
    M.rollBack(C.cursor, C.len) *{Rollback hidden state of M}*
    C.cursor $\leftarrow C$.len $- 1$ *{Update cursor in C}*
  end if
else
  {Directly append the new token to C}
  C.append($newToken$)
end if

introduces memory costs, as it requires storing hidden states. We defer the analysis of this overhead to Section 3.4.

3.3.2 Prediction When Necessary. After getting the new token, the algorithm (line 10 in Algorithm 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.

3.3.3 Model Refinement. Compressed learning supports continuous model refinement, also referred to as online learning in the literature. After a user-defined learning interval (L), the algorithm can update the predictive model using the compressed sequence from that interval, as shown in lines 19–27 of Algorithm 1. The refinement process can also incorporate compressed sequences from prior intervals for training. Rather than introducing a new online learning or incremental learning algorithm, our contribution lies in minimizing data compression overhead when compressing the sequence from the interval through *partial compression*.

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 21) compresses only the uncompressed parts, thereby reducing compression time. The subroutine first extracts out all the new subsequences in C that do not match non-terminal tokens. In Figure 2(c), there are three such subsequences, "x," "ababdayabf," "g" ("S" is the start marker, hence not included). 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 note that simple concatenation can cause wrong compression results, as the subsequences are not actually consecutive. For instance, "x" and "ababdayabf" are not consecutive as "3" exists between them. However, Sequitur could be misled by the concatenated sequence to group the end of a subsequence and the start of another subsequence into a single token. To avoid the issue, we insert distinctive symbols at the end of a subsequence as separators, as illustrated by the triangles in the "Refine Model" row in Figure 2(c). This ensures that applying Sequitur to compress the sequence "x△ ababdayabf▽g" produces the same compressed results as applying it independently to each subsequence ("x," "ababdayabf," "g").

Accuracy-Conscious Lowering (Solution to Issue-3). Lowering is a crucial step in achieving a balanced tradeoff between token granularity and frequency. Compressed sequences using CFG often include large non-terminal tokens, which represent a long sequence of events and appear infrequently in the compressed sequence. These large tokens can hinder RNNs from effectively learning patterns. Lowering transforms the compressed sequence into a more suitable form, enabling RNNs to achieve accuracy comparable to those trained on uncompressed sequences. 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 ($FREQ$) offers a knob to adjust the tradeoff between the compression rate and the frequency of tokens. In the extreme case where the frequency threshold is too large, there will be no non-terminal symbols and our algorithm becomes the same as default learning. In another extreme case where the frequency threshold is too small, the data sequence will be compressed into a very abstract format that contains mainly less-frequent non-terminal symbols. Although a highly compressed sequence may lead to less frequent hidden state updates and yield larger speedups and prediction scopes, it is not friendly for learning and could result in accuracy degradation. During offline training, compressed learning uses binary search to automatically find

the suitable frequency threshold that meets a user-specified accuracy requirement, as detailed in Section 4.

3.4 Algorithm Analysis

In this part, we analyze the computational complexity of compressed learning on RNN inference, and how rollbacks affect the efficiency. The analysis focuses on RNN inference as the goal of compress learning is to improve inference efficiency. We also analyze the computational complexities in RNN training for completeness.

Analysis on Inference. The total time cost of predictions in compressed learning is $\alpha \cdot \gamma$ fraction of the cost of the default RNN inference on the original sequence, where $\alpha = N_c/N_d$ and $\gamma = T_c/T_d$. Here, N_c and N_d are the number of predictions conducted in compressed learning and in default RNN inference, and T_c and T_d are the average times taken by one prediction in the two cases. The time taken for one prediction is the time to execute recurrence function f_W for updating the hidden state and producing outputs.

We claim that $\alpha \cdot \gamma$ must be no greater than one, even in the presence of rollbacks. Formally, we have the following propositions.

Proposition 1. *Compressed learning, even with rollbacks, does no more predictions than the number of input events—that is,* $\alpha <= 1$.

Proof. The correctness is easy to see if we notice that as Algorithm 1 shows, a rollback does not directly trigger a prediction. Predictions are triggered only on line 12 of the algorithm, which is executed at most once for a new event. □

Proposition 2. *The total number of inputs to the predictive model in compressed learning is no greater than the number of input events—that is,* $\gamma <= 1$.

Proof. The only time when the cursor moves backward is at a rollback time. Notice that the movement is to put the cursor right before the new token which is the token at the end of the compressed input. In effect, it just adds one token (i.e., the new token) into the set of inputs possibly sent to the predictive model, as a response to the advent of the new event that triggers the rollback. Therefore, even with rollbacks, the number of inputs to the predictive model is no more than the number of input events. □

As the default run of RNN takes in each input event and makes one prediction per event, the two propositions entail that even in the worst case, neither α nor γ would be greater than one. In practice, because in compressed learning, predictions are invoked only at some events (line 12 in Algorithm 1) and the input to the model is compressed, $\alpha \cdot \gamma$ is typically much smaller than one, leading to a better prediction efficiency. For the same reasons, the refinement of the predictive model in compressed learning also costs no more than the default given the same number of training epochs.

Compressed learning adds extra operations for RNN inference. They are primarily (1) online tokenization prior to prediction, and (2) the recording of hidden states for possible future rollbacks. Tokenization has a linear time complexity in terms of the length of the input sequence. As inference involves a number of matrix multiplications, the time online tokenization takes is marginal relative to the inference time savings, as Section 4 will show. Recording hidden states in memory does not take time as they are already created in memory, but consumes space. The number of hidden states needed to record is bounded by $m = min(L, K)$, where L is the length of a learning interval, and K is the length of the longest token in the vocabulary. If m exceeds memory budget, the implementation can limit it to fit the memory budget, and accordingly, limit rollbacks to tokens shorter than m.

Analysis on Training. Compressed learning introduces a compression overhead prior to training but could significantly reduce the length of sequences used during training. We analyze and compare the time complexities of training in compressed learning with those of the default approach. In the default RNN training approach, uncompressed sequences are used to train the RNN. Let E_d represents the number of epochs required for the model to converge, and T_{epoch} denotes the time required to complete one epoch. The total time to train the model can be expressed as $O(E_d * T_{\text{epoch}})$. Since T_{epoch} = #iterations * $T_{\text{iteration}}$, and #iterations is proportional to the uncompressed sequence length (L_d), the time complexity for training a well-trained model becomes $O(E_d * L_d * T_{\text{iteration}})$.

In compressed learning, the RNN is trained on compressed sequences. The total training time includes stage 1 offline compression and stage 2 offline RNN training. The time complexity for stage 1 is $O(L_d)$, as Sequitur, the compression algorithm used, operates with linear time complexity. For stage 2, let L_c be the length of the compressed sequence, E_c the number of epochs needed to converge, and $T_{\text{iteration}}$ the time per iteration. The time complexity for this stage is $O(E_c * L_c * T_{\text{iteration}})$, assuming the batch size remains the same as the default approach. Thus, the overall time complexity for training an RNN using compressed learning is: $O(L_d) + O(E_c * L_c * T_{\text{iteration}})$.

The compressed sequence length is typically shorter than the uncompressed sequence length (i.e., $L_c < L_d$). As a result, compressed learning—despite accounting for the overhead of data compression—achieves 1.2–39.4 times faster training per epoch compared to the default approach, as demonstrated in Table 5. On the other hand, when training from scratch, more epochs are typically needed to converge on compressed sequences compared to un-compressed sequences (i.e., $E_c > E_d$). Therefore, whether compressed learning achieves faster end-to-end training than the default approach ultimately depends on the specific dataset and its characteristics.

4 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 online tokenization for inference and the partial compression for online model refinement? After introducing the experiment settings in Section 4.1, we answer the first two questions in Section 4.2 and the last question in Section 4.3.

4.1 Experiment Settings

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 2 lists the 16 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 [30], prefetching [4, 16, 51], and system anomaly detection [12].

The system log traces come from LogHub [29]; they are real-world system traces of Microsoft Windows OS and Thunderbird Linux Cluster from Sandia National Lab. Following prior works in log parsing [22, 29], we replaced date, timestamp, package specs, and parameter values in each log entry as a dummy string to convert the unstructured free-text log entries into a sequence of log keys. Each event is a log key, which is also known as message type. The other traces were collected through Intel instrumentation tool (Pin [37]) on six real-world applications.

Table 2. Sequence Statistics

Sequences		Compression ratio (X)	#Non-terminal symbols	Token length stats		
No.	Name			Min	Mean	Max
1	fluid-calls	3,759.4	6	4	1,507.3	8,192
2	go-calls	12.2	436	2	14.3	80
3	molecule-calls	96.0	155	2	78.2	1,024
4	perl-calls	79.8	116	2	88.4	1,880
5	ocean-calls	747.4	27	2	293.7	2,194
6	waves-calls	2,066.1	16	2	1,051.1	8,192
7	fluid-mem	2,487.6	8	2	1,128.9	5,120
8	go-mem	86.2	30	2	339.8	3,216
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	2,048
16	thunderbird-log2	20.9	428	2	17.1	1,536

Every sequence contains 500K events. 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.

Table 3. The Applications and Systems on Which Traces Are Obtained

Name	KLOC	Description
fluid [46]	1	A dynamic fluid simulation algorithm using Lattice Boltzmann Method.
go [18]	21	A go playing engine using Monte Carlo tree search.
molecule [1]	24	A molecular modeling application.
ocean [3]	210	A regional ocean modeling algorithm.
perl [2]	362	Perl interpreter.
wave [56]	1	A 3D wave modeling algorithm.
windows [29]	N/A	Windows 7 event log that keeps track of package installation and updates.
thunderbird [29]	N/A	Thunderbird supercomputer log that contains alert and non-alert messages. Used for alert detection and prediction research [43].

Table 3 lists those applications and the sources. These applications are from various domains, from fluid dynamics to programming language interpreters and stochastic modeling. The regularity of the behaviors of those applications also varies significantly. Program `fluid`, for instance, is very regular; the core of it is structured linear algebra. Program go, on the other hand, as a stochastic tree search through Monte Carlo (random walk), is inherently random. Such a collection allows the experiments to check whether the compressed learning can discover and effectively leverage the repetitive patterns in a trace, and at the same time, avoid negative impact (slowdown, accuracy loss) if it is applied to trace the lack of such regular patterns.

Table 2 also shows the sequence statistics. Each sequence contains 500,000 events. The interval size is 50,000 so one sequence consists of a total of 10 intervals. For each sequence, we use the first five intervals for model training and the rest for model testing (e.g., online prediction). Continuous model refinement is disabled by default. If it is enabled, the learning on a subsequence happens after the prediction on that subsequence is done.

Counterparts for Comparisons. Since 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 *uncompressed 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. All the prior works on applying RNNs to program trace analysis [21, 28, 49, 61] and system log analysis [22] use this strategy.
(2) *Default learning with k-event prediction (default-k).* This method also trains the RNN using the *uncompressed sequence* but has the same prediction scope as our *compressed learning* has. Specifically, after predicting an event, the method feeds the prediction back into the RNN to make the next prediction and continues this process until k consecutive events are predicted, where k is the average length of a prediction in our *compressed learning*. Unlike *compressed learning*, *default-k* predicts the next k events by making k consecutive predictions instead of a single prediction. As a result, it saves no prediction time compared to *default-1*. To clarify, suppose one prediction takes x seconds. Then, k predictions will take $k * x$ seconds. For a sequence of length k, *default-k* requires $k * x$ seconds to complete the prediction tasks. Similarly, *default-1* makes one prediction per event, resulting in k predictions in the same sequence and taking the same $k * x$ seconds to complete the tasks.

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 gated recurrent unit layer with 1,024 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 [33] using an input length of 100 for all methods. The total number of epochs is set to a maximum of 100. Training stops when the loss stop decreases in five epochs. To minimize additional complexity from hyperparameter tuning, we use a straightforward training setup: the batch size is set to one, and the ADAM optimizer is initialized with a learning rate of 0.001. We did not apply L1/L2 regularization or dropout, as the RNN model converges to good accuracy without these enhancements. 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. Specifically, we allow users to specify a tolerable accuracy drop (e.g., 1%) compared to *default-1*. We use the first 4 training intervals for RNN training and the remaining 1 training interval for validation. We increase *FREQ* to lower the compression rate in exchange for better model quality or decrease it for a higher speedup. To reduce the overhead of binary search, the options of *FREQ* are currently limited to 14 values: 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000, 5,000, 10,000, 20,000, and 50,000. When *FREQ* reaches 50,000 which is the interval size, the sequence is not compressed and compressed learning falls back to default-1 so that the same accuracy is guaranteed. If one changes the interval length, the list of options can be adjusted accordingly.

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 (e.g., tokenization, rollback) 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.

Table 4. 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	8,916	522	0.00017	0.132	3.701	28.1	99.74	98.54	99.89
	1	5	133	7,372	1180	0.00018	0.082		45.3	99.57	98.59	
molecule-calls	0	50	30	8,572	722	0.00044	0.179	3.582	20.0	99.67	95.86	99.6
	1	5	91	4,393	1099	0.00287	0.071		50.5	99.5	77.37	
ocean-calls	0	500	17	14,677	191	0.00007	0.209	3.726	17.9	99.89	99.51	99.95
	1	5	200	5,136	725	0.00282	0.084		44.5	98.94	44.59	
wave-calls	0	20	4798	4,070	21	0.00616	0.246	3.707	15.1	83.74	83.76	83.76
	1	20	4798	4,070	21	0.00616	0.246		15.1	83.74	83.76	
go-calls	0	20,000	1	248,732	2132	0.00001	3.608	3.608	1	87.4	87.59	87.59
	1	5000	2	229,512	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	3,336	55	0.00001	0.054	3.611	66.4	98.82	91.49	99.04
	1	5	78	3,336	55	0.00002	0.054		66.4	98.82	91.49	
perl-mem	0	20	81	6,273	105	0.00014	0.086	3.630	42.2	99.48	98.52	98.74
	1	20	81	6,273	105	0.00014	0.086		42.2	99.48	98.52	
ocean-mem	0	20	4	83,383	3640	0.00002	1.478	3.954	2.7	81.63	75.93	81.3
	1	5	6	78,782	1166	0.00004	1.236		3.2	80.69	72.37	
wave-mem	0	500	2	193,050	1902	0.00001	3.084	3.700	1.2	79.32	64.71	79.56
	1	5	5	85,957	1416	0.00001	1.233		3	79.1	50.71	
molecule-mem	0	2,000	1	250,000	0	0.00001	3.625	3.625	1	93.3	93.36	93.36
	1	1,000	2	210,869	1495	0.00000	3.296		1.1	92.05	73.69	
windows-log1	0	200	11	40,930	3768	0.00004	0.605	3.750	6.2	95.13	88.91	95.91
	1	20	31	24,391	6996	0.00017	0.364		10.3	93.25	79.54	
windows-log2	0	100	14	38,781	4636	0.00005	0.716	4.009	5.6	95.37	90.62	96.49
	1	20	23	29,882	9836	0.00012	0.617		6.5	95.08	91.81	
thunderbird-log1	0	1,000	3	109,977	4580	0.00011	1.521	4.014	2.6	93.91	94.02	94.02
	1	200	10	40,933	2445	0.00026	0.628		6.4	92.37	84.93	
thunderbird-log2	0	20,000	1	250,000	0	0.00001	3.531	3.531	1	92.45	92.52	92.52
	1	1,000	2	135,229	11,498	0.00007	1.962		1.8	91.04	83.53	

The table shows the clear benefits from our compressed learning on both prediction scope measured by average prediction length ("Avg. pred Length") and average prediction speed ("Prediction Speedup"). *Avg. latency: averaged time spent on predicting the next event. Default-1 and default-k have the same avg. latency.

Platform and Hardware. All the experiments are performed with TensorFlow 2.2 on computing nodes equipped with a 12-core 2.40 GHz Intel Xeon E5-2620 v3 processor, 256GB of RAM, 256GB SSD and NVIDIA TITAN X GPUs. For each experiment, a memory limit of 60GB and a limit of 1 GPU usage were set. CUDA version is 10.1. We use sequitur implementation available at http://www.sequitur.info/.

4.2 Results on Prediction Scope and Speed

Table 4 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 4 shows the clear benefits of our compressed learning on both prediction scope and speed compared to *default-1*. **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** (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. Figure 3 illustrates the function call sequence and the predictions made by the RNN trained with compressed

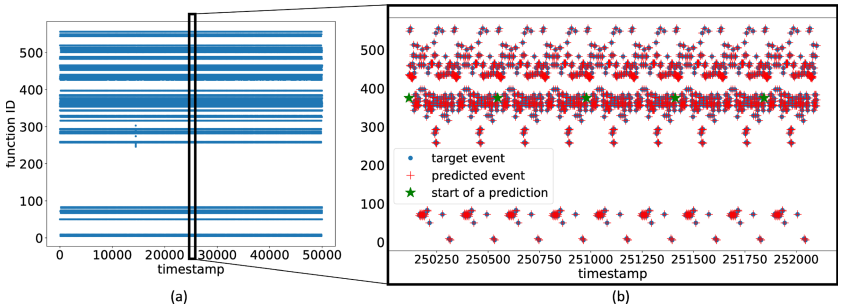


Fig. 3. (a) The function call sequence from `perl` and (b) predictions made by the RNN trained with compressed learning.

learning. For the part of the call sequence of `perl` with regular repetitive patterns, the RNN in compressed learning can recognize the patterns and makes predictions much less frequently.

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, as Section 4.1 has mentioned, to predict k events, *default-k* still needs to make k predictions because it needs to process all the events to update the hidden states; it has the same prediction time as the default-1 approach.

The Effectiveness of the Technique Holds Across Domains and Sequence Types. 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. Overall, 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.

4.3 Runtime Overhead

Rollback Overhead. The myopic nature of online tokenization incurs a number of rollbacks in compressed learning for most sequences, as the "#rollbacks" column in Table 4 shows. But as Section 3.4 has proved, 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.

Runtime Overhead of Tokenization. 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 4. 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.

Runtime Overhead of Partial Compression. In another experiment, we turned on continual model refinement for all three methods in comparison. We did not see noticeable changes in the prediction results (accuracy, speedup, scope). The reason is that the one-epoch retraining is not enough to create some notable improvement in the model quality on these traces. The refinement incurs extra runtime overhead for compressed learning, as it needs to do the partial compression.

Table 5 reports the time overhead of partial compression and also the speedups of our compressed learning on the one-epoch training compared to default approaches (*default*). For compressed learning, we used the FREQ that corresponds to a specified accuracy drop of 1%. *default* refers to both *default-1* and *default-k* because they use the same training strategy (e.g., train on uncompressed sequences) and thus require the same amount of training time. Overall, the time overhead of partial compression is minimal, taking up to 6.08% of the total model refinement time (including the time spent on partial compression). Increasing the number of epochs for model refinement will further reduce the percentage of compression overhead. As a side effect of compression, continual model refinement on compressed sequences for one epoch is up to 39.4× faster than default approaches.

Table 5. Runtime Overhead of Continual Model Refinement

Sequences		Compression overhead (%)	Refinement speedup (×)
No.	Name		
1	fluid-calls	0.03	22.1
2	go-calls	4.35	1.2
3	molecule-calls	0.06	5.4
4	perl-calls	0.02	11
5	ocean-calls	0.03	7.8
6	waves-calls	2.04	21.5
7	fluid-mem	0.1	39.4
8	go-mem	0.16	7.6
9	molecule-mem	6.08	1.8
10	ocean-mem	0.96	4.4
11	perl-mem	1.29	5.9
12	waves-mem	2.12	5
13	windows-log1	0.01	5.6
14	windows-log2	0.01	5.9
15	thunderbird-log1	0.02	3.4
16	thunderbird-log2	0.03	1.7

Each RNN model is refined for one epoch for both *default-1* and compressed learning.

5 Related Work

Deep Learning on Compressed Inputs. There are some studies on DNN training and inference with compressed input data, but all on images and convolutional neural networks [24, 27, 36, 54, 60]. In Natural Language Processing, the representation of inputs sometimes uses some tokens to represent some common phrases. An example is Byte Pair Encoding [25] 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 [49] uses an offline attention-based **Long Short-Term Memory (LSTM)** model to provide insights for designing a simple online hardware cache replacement policy. Perceptron-based Prefetch Filtering [7] enhances the existing state-of-the-art prefetchers by observing the stream of candidate prefetches generated by a prefetcher, and then rejects those that are predicted by the online-trained neural model to be inaccurate. Another study [28] 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 [21] 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. System logs record detailed software runtime information, which allows software developers to track and analyze system behaviors. Recent years have seen a growing interest in applying Deep Learning models in analyzing system logs. One study [22] proposes Deeplog, which leverages LSTM for online anomaly detection. Another work [11] proposes to use RNN with the attention mechanism for anomaly detection. Some other work [40] builds an RNN-based content caching framework to predict the popularity of content objects on information-content networks. Wang and others [57] 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.

Online Learning, Incremental Learning, and Curriculum Learning. We discuss a few concepts that are related to compressed learning. Online learning [48] is a machine learning paradigm where the model learns incrementally by processing data one instance (or a small batch of instances) at a time as it becomes available. Unlike offline (or batch) learning, which trains the model on the entire dataset before making predictions, online learning updates the model continuously based on new data, enabling real-time adaptation. In the context of compressed learning, Stage 2 (offline RNN training) corresponds to offline learning, while the optional model refinement in Stage 3 represents online learning. Rather than introducing new online learning algorithms, this work demonstrates that online learning can be seamlessly integrated into the proposed approach. It allows models to train on compressed sequences instead of uncompressed ones, with minimal compression overhead. Incremental learning (or continuous learning) [17, 44] processes data sequentially over time but focuses on avoiding catastrophic forgetting while adapting to new information. Incremental learning techniques can be incorporated into the optional model refinement step of the proposed approach, ensuring the model retains previously learned knowledge while adapting to new compressed sequences instead of uncompressed sequences.

Curriculum Learning [5] is a training strategy inspired by the way humans learn, where models are trained on tasks or data in a meaningful order, starting with simpler examples and gradually progressing to more complex ones. This approach leverages the idea that mastering easier concepts first helps the model build a strong foundation, making it more capable of handling challenging concepts later. In the context of the proposed approach, curriculum learning could be integrated by carefully selecting and ordering compressed data sequences to train the RNN.

Sequitur [42] has been applied to various tasks, including program and data pattern analysis [15, 20, 34]. It has not been introduced into RNN learning before.

6 Discussions and Limitations

Tradeoffs between Compression Rate, Accuracy, and Efficiency. A key aspect of compressed learning is the inherent tradeoff between compression rate, accuracy, and inference efficiency. On the one hand, more aggressive compression (i.e., using lower frequency thresholds during lowering, see Section 3.3.3) shortens the effective sequence length, thereby reducing the number of RNN invocations and significantly improving inference speed and prediction scope. On the other hand, such aggressive compression may produce tokens that occur too infrequently, making it harder for the RNN to reliably learn their patterns and resulting in accuracy degradation. Conversely, less aggressive compression (i.e., higher frequency thresholds) mitigates accuracy drops, but at the cost of reduced compression benefits and longer inference times. This tradeoff is directly observed in our experimental results (Table 4). For example, in the *ocean-calls* sequence, a frequency threshold of 5 yields an average prediction length of 200 events with a speedup of 44.5×, but also a modest accuracy drop of about 1%. By contrast, a higher threshold of 500 results in shorter predictions (17 events on average), smaller speedup (17.9×), but essentially no accuracy loss. Similar patterns

hold across other datasets, demonstrating that compressed learning offers an explicit knob for practitioners to tune according to their latency and accuracy requirements. Overall, users can select the frequency threshold *FREQ* to achieve a desired balance among accuracy, inference speed, and prediction scope.

Relation to Transformer-based Models. Transformers [55] have been proposed to enable longer-context modeling. These architectures provide strong baselines in many natural language tasks, but they often require large-scale training data and incur higher inference overheads than RNNs in domains such as memory traces and system logs, where real-time prediction is critical. Our work is complementary to these advances: rather than modifying model architectures, we focus on compressing the input sequence itself, which can in principle also benefit Transformer inference by reducing sequence length. Exploring compressed learning as a preprocessing step for Transformers is an interesting direction for future work.

Limitations on RNNs. While our work demonstrates that compressed learning substantially improves both the efficiency and scope of RNN inference in practice, it is important to acknowledge the theoretical limitations of RNNs with respect to formal language recognition. RNNs with finite precision and finite computation time are not expressive enough to fully recognize CFGs. Although RNNs are Turing complete in principle, such expressiveness requires unrealistic assumptions of infinite precision and unbounded computation time [50, 58]. Recent studies have formalized these limitations, showing that sequential neural networks behave as finite-state automata [39], and that practical neural architectures face significant expressiveness constraints [19, 52].

These limitations imply that our method does not enable RNNs to fundamentally recognize arbitrary CFGs. Rather, compressed learning leverages grammar-based compression to reshape the input sequences into more compact representations that highlight repetitive hierarchical structures. This transformation allows RNNs to process fewer steps and make longer-range predictions, improving inference efficiency and expanding prediction scope, but it does not overcome the theoretical expressiveness ceiling of RNNs. In other words, the benefits reported in this article come from exploiting structural regularities in practical data sequences, not from extending the formal recognition power of RNNs.

7 Conclusion

This article presents *compressed learning* that integrates 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. While our study focuses on improving the efficiency of RNN inference, we note that compression-based techniques may also complement efficient Transformer variants. Since both model families face tradeoffs between context length and inference efficiency, integrating grammar-based compression with Transformer architectures is a promising avenue for future research.

References

[1] AmberMD. 2020. Molecular Modeling. Retrieved September 9, 2020 from http://ambermd.org/
[2] Perl.Org. 2020. Perl Interpreter. Retrieved September 9, 2020 from http://www.perl.org/
[3] ROMS. 2020. Regional Ocean Modeling System. Retrieved September 9, 2020 from http://www.myroms.org/
[4] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying memory access patterns for prefetching. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 513–526.

[5] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 41–48.

[6] Alex Beutel, Paul Covington, Sagar Jain, Can Xu, Jia Li, Vince Gatto, and Ed. H. Chi. 2018. Latent cross: Making use of context in recurrent recommender systems. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 46–54.

[7] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-based prefetch filtering. In *Proceedings of the 46th International Symposium on Computer Architecture*.

[8] Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. 2015. Finger search in grammar-compressed strings. arXiv:1507.02853. Retrieved from https://arxiv.org/abs/1507.02853

[9] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. 2015. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing* 44, 3 (2015), 513–539.

[10] Nieves R. Brisaboa, Adrián Gómez-Brandón, Gonzalo Navarro, and José R. Paramá. 2019. Gract: A grammar-based compressed index for trajectory data. *Information Sciences* 483 (2019), 106–135.

[11] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. 2018. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *Proceedings of the 1st Workshop on Machine Learning for Computing Systems*, 1–8.

[12] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Computing Surveys* 41, 3 (2009), 1–58.

[13] Haibin Cheng, Pang-Ning Tan, Jing Gao, and Jerry Scripps. 2006. Multistep-ahead time series prediction. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 765–774.

[14] Lin Cheng, Yuliang Shi, Kun Zhang, Xinjun Wang, and Zhiyong Chen. 2021. GGATB-LSTM: Grouping and global attention-based time-aware bidirectional LSTM medical treatment behavior prediction. *ACM Transactions on Knowledge Discovery from Data* 15, 3 (2021), 1–16.

[15] Trishul M. Chilimbi. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. *ACM SIGPLAN Notices*, 36 (2001), 191–202.

[16] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. IEEE, 14–25.

[17] Andrea Cossu, Antonio Carta, Vincenzo Lomonaco, and Davide Bacciu. 2021. Continual learning for recurrent neural networks: An empirical evaluation. *Neural Networks* 143 (2021), 607–627.

[18] Rémi Coulom. 2007. Computing "ELO ratings" of move patterns in the game of go. *ICGA Journal* 30, 4 (2007), 198–208.

[19] Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. 2023. Neural networks and the Chomsky hierarchy. In *Proceedings of the 11th International Conference on Learning Representations*.

[20] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2015. Using formal grammars to predict I/O behaviors in HPC: The omnisc'IO approach. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2015), 2435–2449.

[21] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, 37–48.

[22] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 1285–1298.

[23] Jeffrey L. Elman. 1990. Finding structure in time. *Cognitive Science* 14, 2 (1990), 179–211.

[24] Dan Fu and Gabriel Guimaraes. 2016. Using Compression to Speed Up Image Classification in Artificial Neural Networks. Retrieved from https://danfu.org/files/CompressionImageClassification.pdf

[25] Philip Gage. 1994. A new algorithm for data compression. *C Users Journal* 12, 2 (1994), 23–38.

[26] Hui Guan, Umang Chaudhary, Yuanchao Xu, Lin Ning, Lijun Zhang, and Xipeng Shen. 2021. Recurrent neural networks meet context-free grammar: Two birds with one stone. In *Proceedings of the 2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1078–1083.

[27] Lionel Gueguen, Alex Sergeev, Ben Kadlec, Rosanne Liu, and Jason Yosinski. 2018. Faster neural networks straight from jpeg. In *Advances in Neural Information Processing Systems*, Vol. 31, 3933–3944.

[28] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *Proceedings of the International Conference on Machine Learning*.

[29] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2020. Loghub: A large collection of system log datasets towards automated log analytics. arXiv:2008.06448. Retrieved from https://arxiv.org/abs/2008.06448

[30] Jose A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N. Patt. 2008. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 80–90.

[31] Alain Ketterlin and Philippe Clauss. 2008. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 94–103.

[32] Minje Kim. 2023. *A Page Scheduler Using Machine Learning for Hybrid Memory Systems*. Master's thesis. Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea. Retrieved from https://scholarworks.unist.ac.kr/handle/201301/73941

[33] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. *Proceedings of the International Conference on Learning Representations*.

[34] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. 2005. Motivation for variable length intervals and hierarchical phase behavior. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '05)*. IEEE, 135–146.

[35] Liu Liu, Lei Deng, Zhaodong Chen, Yuke Wang, Shuangchen Li, Jingwei Zhang, Yihua Yang, Zhenyu Gu, Yufei Ding, and Yuan Xie. 2020. Boosting deep neural network efficiency with dual-module inference. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 6205–6215.

[36] Zihao Liu, Tao Liu, Wujie Wen, Lei Jiang, Jie Xu, Yanzhi Wang, and Gang Quan. 2018. DeepN-JPEG: A deep neural network favorable JPEG-based image compression framework. In *Proceedings of the 55th Annual Design Automation Conference*.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (2005), 190–200.

[38] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*. PMLR, 4505–4515.

[39] William Merrill. 2019. Sequential neural networks as automata. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*. Jason Eisner, Matthias Gallé, Jeffrey Heinz, Ariadna Quattoni, and Guillaume Rabusseau (Eds.), Association for Computational Linguistics, 1–13. DOI: https://doi.org/10.18653/v1/W19-3901

[40] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 48–53.

[41] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. 2016. Phased LSTM: Accelerating recurrent network training for long or event-based sequences. In *Advances in Neural Information Processing Systems*, Vol. 29, 3882–3890.

[42] C. G. Nevill-Manning and I. H. Witten. 1997. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* 7 (1997), 67–82.

[43] Adam Oliner and Jon Stearley. 2007. What supercomputers say: A study of five system logs. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*. IEEE, 575–584.

[44] Alexander Ororbia, Ankur Mali, C. Lee Giles, and Daniel Kifer. 2020. Continual learning of recurrent neural networks by locally aligning distributed representations. *IEEE Transactions on Neural Networks and Learning Systems* 31, 10 (2020), 4267–4278.

[45] Seong Hyeon Park, ByeongDo Kim, Chang Mook Kang, Chung Choo Chung, and Jun Won Choi. 2018. Sequence-to-sequence prediction of vehicle trajectory via LSTM encoder-decoder architecture. In *Proceedings of the 2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1672–1678.

[46] Yue-Hong Qian, Dominique d'Humières, and Pierre Lallemand. 1992. Lattice BGK models for Navier-Stokes equation. *Europhysics Letters* 17, 6 (1992), 479.

[47] Deepika Saxena, Jitendra Kumar, Ashutosh Kumar Singh, and Stefan Schmid. 2023. Performance analysis of machine learning centered workload prediction models for cloud. *IEEE Transactions on Parallel and Distributed Systems* 34, 4 (2023), 1313–1330.

[48] Shai Shalev-Shwartz. 2012. Online learning and online convex optimization. *Foundations and Trends in Machine Learning* 4, 2 (2012), 107–194.

[49] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[50] Hava T. Siegelmann and Eduardo D. Sontag. 1992. On the computational power of neural nets. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory*. 440–449.

[51] Yan Solihin, Jaejin Lee, and Josep Torrellas. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE, 171–182.

[52] John Stogin, Ankur Mali, and C. Lee Giles. 2024. A provably stable neural network Turing machine with finite precision and time. *Information Sciences* 658, C (Feb. 2024), 120034. DOI: https://doi.org/10.1016/j.ins.2023.120034
[53] Fu Jie Tey, Tin-Yu Wu, and Jiann-Liang Chen. 2022. Machine learning-based short-term rainfall prediction from sky data. *ACM Transactions on Knowledge Discovery from Data* 16, 6 (2022), 1–18.
[54] Róbert Torfason, Fabian Mentzer, Eiríkur Ágústsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. 2018. Towards image understanding from deep compression without decoding. In *Proceedings of the International Conference on Learning Representations*.
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. arXiv:1706.03762. Retrieved from https://arxiv.org/abs/1706.03762
[56] Arsi Vaziri and Mark Kremenetsky. 1995. *Visualization and Tracking of Parallel CFD Simulations*. Technical Report NAS-95-004. NASA Ames Research Center, Numerical Aerodynamic Simulation Systems Division, Moffett Field, CA. Retrieved from https://www.nas.nasa.gov/assets/nas/pdf/techreports/1995/nas-95-004.pdf
[57] Hanson Wang, Zehui Wang, and Yuanyuan Ma. 2019. Predictive precompute with recurrent neural networks. arXiv:1912.06779. Retrieved from https://arxiv.org/abs/1912.06779
[58] Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. On the practical computational power of finite precision RNNs for language recognition. arXiv:1805.04908. Retrieved from https://arxiv.org/abs/1805.04908
[59] Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-sequence learning as beam-search optimization. arXiv:1606.02960. Retrieved from https://arxiv.org/abs/1606.02960
[60] Xiufeng Xie and Kyu-Han Kim. 2019. Source compression with bounded DNN perception loss for IoT edge computer vision. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*.
[61] Yuan Zeng and Xiaochen Guo. 2017. Long short term memory based hardware prefetcher: A case study. In *Proceedings of the International Symposium on Memory Systems*, 305–311.
[62] Matthew Shunshi Zhang and Bradly Stadie. 2019. One-shot pruning of recurrent neural networks by Jacobian spectrum evaluation. arXiv:1912.00120. Retrieved from https://arxiv.org/abs/1912.00120
[63] Maohua Zhu, Jason Clemons, Jeff Pool, Minsoo Rhu, Stephen W. Keckler, and Yuan Xie. 2018. Structurally sparsified backward propagation for faster long short-term memory training. arXiv:1806.00512. Retrieved from https://arxiv.org/abs/1806.00512

Received 25 February 2024; revised 27 September 2025; accepted 6 December 2025