Dynamic Cell Structure via Recursive-Recurrent Neural Networks

Xin Qian[1], Matthew Kennedy[2], and Diego Klabjan[3]

[1,3]Department of Industrial Engineering and Management Sciences, Northwestern University
[2]Weinberg College of Arts and Sciences, Northwestern University

Abstract

In a recurrent setting, conventional approaches to neural architecture search find and fix a general model for all data samples and time steps. We propose a novel algorithm that can dynamically search for the structure of cells in a recurrent neural network model. Based on a combination of recurrent and recursive neural networks, our algorithm is able to construct customized cell structures for each data sample and time step, allowing for a more efficient architecture search than existing models. Experiments on three common datasets show that the algorithm discovers high-performance cell architectures and achieves better prediction accuracy compared to the GRU structure for language modelling and sentiment analysis.

1 Introduction

First proposed by Hopfield [12], recurrent Neural Network (RNN) models excel at machine learning tasks that involve sequential data such as natural language processing. Researchers soon noted that a major obstacle of RNN models is in backpropagation when computing gradients. Since RNNs are trained by backpropagation through time, when the recurrent structure is unfolded into a huge feed-forward network with many layers, gradients tend to grow or vanish exponentially in the same way as in very deep feed-forward neural networks [20]. Many extensions of RNN models, such as Long Short-Term Memory (LSTM) [11] and Gated Recurrent Units (GRU) [5], are proposed to address this problem. These models achieve state-of-the-art results in many machine learning tasks like language modeling [6] and speech recognition [15, 22].

However, the cell structure of these hand-crafted RNN models, like LSTM and GRU, is fixed across all time steps and data samples. It is also a time-consuming and tedious effort to find a suitable cell structure through trial and error [19]. Lastly, there is no universal answer to which cell structure to use when facing different types of data and a different problem at hand. Therefore, a more flexible model that can automatically determine the cell structures based on a finite set of trainable parameters is needed to deal with more and more complicated and diversified data sources and problems.

There is another line of research about Recursive Neural Network (RecNN) models [24]. A RecNN model is defined over recursive tree structures – each node of the tree corresponds to a vector computed from its child nodes, and the information passes from the leaf nodes and internal nodes to the root node in a bottom-up manner. The model produces a structured prediction such as a tree by applying the same set of trainable parameters recursively. Derivatives of errors are computed with back-propagation over the tree structures [8]. RecNN has shown great success in learning tree structures of certain natural language processing tasks [25] because the structures it dynamically produces are customized for each data sample.

We consider how to replace the cell structure in RNN models to be time-variant and sample-dependent. We note that the equations governing a cell can be represented as a computational tree where each non-leaf node corresponds to a vector that is computed from the vectors on its two child nodes. The

initial multiset of vectors is composed of the current feature vector at time t and all vectors produced by the previous cell (hidden state representation). If we augment this multiset with constant vectors, such as the zero vector, we can then express mathematical equations behind a cell as a tree on this multiset. RecNN is an appropriate model to capture such a tree by means of a finite set of trainable parameters. In summary, our proposed model is using RecNN in each time step as a replacement for a fixed set of equations. In this way we obtain an architecture with cells depending on time and on each individual sample. In addition to this flexibility, the approach does not require hand-crafting of cells.

Our model shows great results on a series of language modeling and sentiment prediction tasks. In the experiments we show that RRNN is able to design sample-dependent tree structures on the Wikipedia dataset and achieves 5.5% improvement in Bits per Character (BPC) compared to GRU. The performance on the datasets also show the advantage of dynamically designing cell structures for each sample.

The major contribution of this paper is a novel architecture that dynamically searches for the structure of cells in an RNN. Our model, called a Recursive-Recurrent Neural Network (RRNN), recursively designs the cell structure with the help of a scoring function and allows us to build different cell structures under a fixed set of parameters. The proposed model can generate the cell structure of some traditional RNN models, like GRU and LSTM which we establish theoretically. Most importantly, the output tree structure of hidden cells in RRNN are customized based on each data sample, and therefore they are time-variant and data-dependent. Besides, we define a new tree distance metric that can measure the difference between the tree with vectors on each of its nodes. We also exhibit and prove the sufficient and necessary conditions for avoiding the gradient exploding and vanishing problem that usually appears in recurrent neural network models. While such results are known for RNNs, they have not yet been established for RecNNs. Furthermore, our result applies to RRNNs which are a combination of RNN and RecNN.

The rest of the manuscript is structured as follows. In Section 2 we review the literature while in Section 3 we present the RRNN model, including an algorithm to construct trees, the design of the loss function, and other extensions. Section 4 presents some properties of the RRNN model. In Section 5 we introduce the data sets and discuss all experimental results. We defer the proofs of the theorems and other technical details to Appendix.

2 Literature Review

A recurrently connected structure in RNN can improve the performance of a model by its ability to infer sequential dependencies [16]. Despite their success, vanilla RNN models are still limited by the algorithms employed due to the problems of exploding or vanishing gradients that may appear in the training phase [2]. LSTM [11] is one of the most popular ways to address this problem. Many variants are then proposed to improve the performance of LSTM [10, 14]. RNN models often work well if a hand-crafted cell structure is well-designed, which requires time and expertise, and it leads to a fragile setting that works only on a particular problem or, worse, on a single dataset. This is clearly less general and less flexible than the method proposed in this paper where the cells are algorithmically designed.

Recursion is the division of a problem into subproblems of the same type and the application of an algorithm to each subproblem. It can help with augmenting neural architectures and improving the generalization ability of a model [3]. RecNN greedily searches hierarchical tree structures and achieves state-of-the-art performances on tasks like semantic analysis in natural language processing and image segmentation [24, 25].

To provide better flexibility and robustness, automatically searching a neural network architecture is thus a logical next step. Neural Architecture Search (NAS), a subfield of AutoML, is a method which algorithmically finds an architecture; it has significant overlap with hyper-parameter optimization and meta-learning [7]. A simple approach to NAS is to build a layer-chained neural network where layers are differentiated by their choices of operations (pooling, convolution, etc.), activation functions (ReLU, Sigmoid, etc.), width, etc. [4, 26, 1]. Despite its impressive empirical performance, NAS is computationally expensive and time consuming [29].

Various methods of producing novel cell structures for RNNs have been recently proposed. [28] introduce a reinforcement learning approach that utilizes policy gradient to search for convolutional

and recurrent neural architectures. However, the reinforcement learning approach is computationally expensive in the sense that obtaining an architecture with state-of-the-art performance on CIFAR-10 and ImageNet requires 1,800 GPU days [29]. [21] accelerate the search process by sharing parameters among potential architectures. [23] introduce a more flexible algorithm that searches for novel RNNs of arbitrary depth and width. [17] relax the discrete architecture space by continuous probability vectors and utilize a gradient based optimization method to derive an optimal architecture. All these methods are extremely computationally demanding and they yield a fixed network architecture for all times and samples. Some exceptions are in [9] and [27] where the proposed models automatically adjust the number of layers of the LSTM model based on time and sample but the cell structures are static. Our RRNN model further extends this property such that the predicted cell structures are time-variant and sample-dependent.

3 Recursive-Recurrent Neural Network Model

Generally, RNNs consist of two parts which are a hidden cell (recurrent cell) and an output layer. A single sample input of an RNN is a sequence of vectors $\{x_t \in \mathbb{R}^p : t = 1, 2, \ldots, T\}$, labeled by time step t. Given a hidden state h_{t-1}, the t-th recurrent cell defines the next hidden state h_t by $h_t = f(x_t, h_{t-1})$. The output layer is usually a simple network that takes x_t and h_t as input and returns $q_t = g(x_t, h_t; \Gamma)$ as output. These two equations are applied for $t = 1, 2, \ldots, T$.

Function f defined above is time-invariant and thus remains the same in all time steps and for all samples. To address this shortcoming, we propose a new model that can dynamically design the recurrent cell structure (i.e. generate different functions f) with respect to the argument vectors. This is inspired by the idea of RecNNs, thus we call it the Recursive-Recurrent Neural Network model. A dynamic architecture has two advantages: (1) no need to hand-craft a cell, and (2) it automatically adjusts based on timestep and sample.

A simple RecNN model starts with a set of input nodes $\{p_1, \ldots, p_n\}$ with corresponding embedding vectors $\{c_1, \ldots, c_n\}$. Two nodes are merged into a parent node using a pair of weight matrices L and R, a bias vector b, and an activation function σ that provides non-linearity. For two nodes p_i and p_j, their parent, denoted by $p_{i,j}$, is also a node with the embedding vector calculated by $c_{i,j} = \sigma\left(Lc_i + Rc_j + b\right)$. In each iteration, we compute the scores $s_{i,j} = W^{\text{score}} c_{i,j}$ for all pairs of nodes (p_i, p_j) and select the pair of nodes (p_{i_1}, p_{j_1}) with the highest score. We next merge nodes p_{i_1} and p_{j_1} into the parent node p_{i_1,j_1} and remove the two child nodes p_{i_1} and p_{j_1} from further consideration. This procedure repeats until all nodes are merged and only one parent node p_{out} remains. The set of parameters and activation function $\{L, R, b, \sigma, W^{\text{score}}\}$ are shared across the whole network. The RecNN model returns p_{out} and the binary tree rooted at p_{out} as the model output.

The RRNN model replaces the fixed hidden cell of RNN by a recursive tree, dynamically determined by an algorithm similar to RecNN. Note that, even with the fixed set of parameters and activation function $\{L, R, b, \sigma, W^{\text{score}}\}$, the RecNN model can dynamically produce different tree structures based on input nodes (vectors). Therefore, in RRNN, the recurrent cell is different across all time steps and data points. We further discuss the RRNN model in the following sections.

3.1 Recursive-Recurrent Neural Network Model Framework

We start with an example of how to represent the hidden cell structure of GRU to be a binary tree with computational information on it. In the following we assume X is a given sample, where $X = (x_1, \ldots, x_T), x_i \in \mathbb{R}^p$ is a sequence of input vectors. Recall that the GRU equations are:

$$r_t = \sigma\left(W_r x_t + W_r' h_{t-1} + b_r\right), \tag{1}$$

$$z_t = \sigma\left(W_z x_t + W_z' h_{t-1} + b_z\right), \tag{2}$$

$$\widetilde{h}_t = \tanh\left(W_h x_t + W_h'\left(r_t \odot h_{t-1}\right) + b_h\right), \tag{3}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h}_t, \tag{4}$$

where $\{W_r, W_r', W_z, W_z', W_h, W_h'\}$ and $\{b_r, b_z, b_h\}$ are parameter matrices and bias vectors of GRU, respectively. Equations (1) – (4) jointly define the function f of the t-th hidden cell of GRU. As shown in Figure 1, the above equations can also be regarded as a binary tree where each node of the tree corresponds to a 3-tuple (binary operator, activation function, bias vector), and each edge is associated with a trainable matrix or identity matrix.

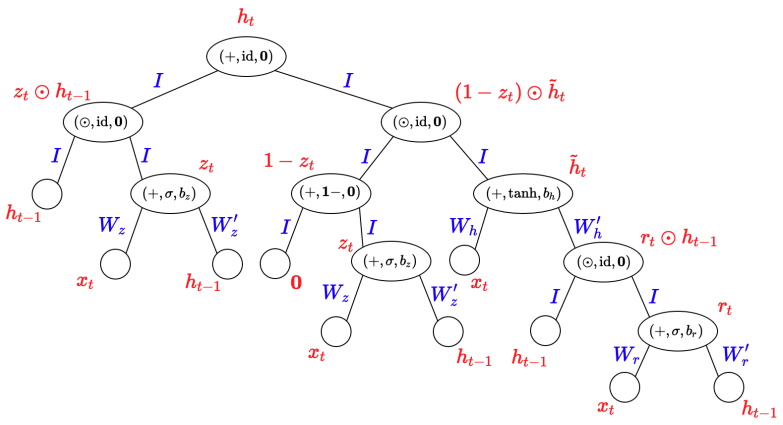


Figure 1: Representation of GRU equations as a binary computational tree. Here red labels are the vectors correspond to each node, blue labels are the weight matrices corresponding to each edge from a child node to its parent, and the 3-tuple in each node is the binary operation, activation function, and the bias vector that are used to calculate the vector of this node. $\mathbf{0}$ stands for the zero vector, $\mathbf{1}-$ the one-minus activation and id the identity activation (mapping).

The tree structure can be achieved in the scheme of RecNN by giving a multiset of initial nodes $\mathcal{N}_0 = \{x_t, x_t, x_t, x_t, h_{t-1}, h_{t-1}, h_{t-1}, h_{t-1}, h_{t-1}, \mathbf{0}\}$. Assuming an appropriate scoring function, in the first iteration, we can find that the parent node that combines x_t and h_{t-1} with parameters and operations $(W_r, W'_r, b_r, +, \sigma)$ has the highest score, thus we merge two nodes x_t and h_{t-1} together to achieve node q_t. In the second iteration, we find that the parent node $z_t = \sigma\left(W_z x_t + W'_z h_{t-1} + b_z\right)$ has the highest score among all potential parent nodes, thus we again take two nodes x_t and h_{t-1} from the node set and merge them to be z_t. After 9 iterations, we end up with one node h_t and this is exactly the output of the t-th hidden cell of GRU. We can prove that, with an appropriate choice of the scoring function, a RecNN can find the tree in Figure 1 and thus it can produce GRU. The statement is given in Section 4.1 and the proof is in the Appendix.

Next, we present the RRNN model. The RRNN model has the same recurrent structure as RNN and an algorithm for building the hidden cell. We start with the case where only one hidden state needs to be transferred between two consecutive hidden cells (GRU falls in this case, but in LSTM we have two states, i.e. the hidden state h_t and the memory state). We extend the model to be compatible with multiple hidden states in Appendix B.

For each hidden cell of RRNN, we build up a binary tree from a multiset of initial nodes with corresponding vectors on each node. A set of trainable parameter matrices and bias vectors, activation functions, binary operations, and a scoring function is given prior to the construction of the tree. We denote $\mathcal{L} = \{L_1, \ldots, L_{n_l}\}$ and $\mathcal{R} = \{R_1, \ldots, R_{n_l}\}$ with $L_i, R_i \in \mathbb{R}^{p \times p}$ as the set of trainable weight matrices, $\mathcal{B} = \{b_1, \ldots, b_{n_l}\}$ with $b_i \in \mathbb{R}^p$ as the set of trainable bias vectors, $\mathcal{U} = \{u_1, u_2, \ldots, u_{n_u}\}$ and $\mathcal{O} = \{o_1, o_2, \ldots, o_{n_o}\}$ as the set of available activation (unary) functions and binary operations, respectively. Besides, a scoring function $\alpha(\cdot; \Theta)$, depending on a set of trainable parameters Θ, is given. A multiset of initial nodes $\mathcal{N}_0 = \{c_1, c_2, \ldots, c_N\}$ with $c_i \in \mathbb{R}^p$ is also given as input of the hidden cell. We do not distinguish between a node and its corresponding vector in the following discussion, however we note that nodes in the tree are unique while the corresponding vectors form a multiset. The tree shown in Figure 1 is the equivalent RRNN representation for the GRU cell.

Formally, the RRNN hidden cell can be understood as a function $f : \mathcal{N}_0 \to (\mathcal{T}, \mathbb{R}^p)$, where $\mathcal{T}$ is the set of all possible (binary) computational trees such that each node of the tree corresponds to a vector and a 3-tuple $(u, o, b), u \in \mathcal{U}, o \in \mathcal{O}, b \in \mathcal{B}$, and each directed edge from each one of the two child nodes to its parent node is associated with a weight matrix.

Function f can be recursively defined as

$$f = f_N \circ f_{N-1} \circ \cdots \circ f_2 \circ f_1, \tag{5}$$

where $f_k : \mathcal{N}_{k-1} \mapsto \mathcal{N}_k, k = 1, 2, \ldots, N-1$ and $f_N : \mathcal{N}_{N-1} \to (\mathcal{T}, \mathbb{R}^p)$. For $k = 1, \ldots, N-1$, function f_k maps multiset $\mathcal{N}_{k-1}$ to multiset $\mathcal{N}_k$ by the following three steps: (i) $C_k = \{c : c = u(o(Lc_i, Rc_j) + b), c_i, c_j \in \mathcal{N}_{k-1}, i < j, L \in \mathcal{L}, R \in \mathcal{R}, b \in \mathcal{B}, u \in \mathcal{U}, o \in \mathcal{O}\}$, (ii) $c_k^* = \arg\max_c \{\alpha(c; \Theta) : c \in C_k\}$, (iii) $\mathcal{N}_k = \{c_k^*\} \cup \mathcal{N}_{k-1} \setminus \{c_i^*, c_j^*\}$, where c_i^*, c_j^* are the two child nodes combined to get c_k^*.

Note that $\mathcal{N}_{N-1} = f_{N-1} \circ \cdots \circ f_1(\mathcal{N}_0)$ contains only one node, i.e. $\mathcal{N}_{N-1} = \{c_{N-1}^*\}$. Function f_N then takes c_{N-1}^* and returns the tree rooted at c_{N-1}^* (we can discover it by unfolding the collapsing decisions and tracing each parent node down to its child nodes until all initial nodes appear) and the corresponding vector $c_{N-1}^* \in \mathbb{R}^p$ as the output. We point out that by definition the produced binary tree is full, i.e. each node has exactly 2 or 0 child nodes.

We next specify the recursive relationship of our cells. To this end, let multiset $\mathcal{N}_0^t = \mathcal{N}_0^t(x_t, h_{t-1})$ consist of several copies of x_t, several copies of h_{t-1} and other constant vectors such as the vector of all zeros or all ones or unit vectors. The numbers of each of them can vary by t. The transition equations and cell output are as follows:

$$(T_t^{\text{pred}}, h_t) = f(\mathcal{N}_0^t(x_t, h_{t-1})), \quad q_t = g(x_t, h_t; \Gamma).$$

It remains to specify the loss function. The generic function is as follows with further details provided in Section 3.3. We assume that a sample consists of (X, Y) where $Y = (y_1, \ldots, y_T)$ is a sequence of ground truth labels. We also assume that we are given a ground truth binary tree T_t^{target} which is specified as in Figure 1 but without the trainable matrices and bias vectors. The target tree usually does not depend on t. Ideally this target tree should not be specified but we leave this as future research work.

One further complication is the fact that the ground truth tree does not have a unique representation. Indeed, since the leaf nodes corresponding to $\mathcal{N}_0$ are unordered, there are several isomorphisms of a given tree that yield the same underlying ground truth transition function, i.e. mathematically equivalent expressions. To this end, let $\text{Iso}(T_t^{\text{target}})$ be the set of all isomorphic trees to T_t^{target}. Note that we do not need to consider the isomorphisms when leaf nodes are ordered, as is the case in [24].

The set of all trainable parameters in RRNN is denoted by $\Phi = \{\mathcal{L}, \mathcal{R}, \mathcal{B}, \Theta, \Gamma\}$. The loss function is specified by

$$L(\Phi) = \mathbb{E}_{(X,Y)} \left[\sum_{t=1}^{T} \left\{ \lambda_1 l(y_t, q_t) + \lambda_2 \min_{\bar{T} \in \text{Iso}(T_t^{\text{target}})} \text{TD}(\bar{T}, T_t^{\text{pred}}) + \lambda_3 \sum_{k=0}^{N-1} m(\mathcal{N}_k^t) \right\} \right] + \lambda_4 \sum_{\phi \in \Phi} \|\phi\|^2 , \tag{6}$$

where function l is the standard loss function, TD measures the difference of two trees, and m is the margin function. These two are described in detail in Section 3.3. The *minimum* operation over isomorphic target trees can also be replaced by expectation.

3.2 Cell Tree Construction

Several changes of constructing the cell tree are made for practical concerns. Functions $f_k, k = 1, \ldots, N-1$ can be regarded as $N-1$ iterations of merging two nodes (vectors). Multisets $\mathcal{N}_k$ can have multiple copies but in practice we keep a single copy that is reused. The new set $\mathcal{N}_{k-1}$ consists of three fixed sets of vectors, namely $\mathcal{S}_t^{\text{data}} = \{x_t\}$ as the set of vectors from the data samples, $\mathcal{S}_t^{\text{prev}}$ as the set of vectors from the previous hidden cell, and $\mathcal{S}_t^{\text{aux}}$ as the set of auxiliary vectors such as the zero vector, etc., together with the set $\mathcal{P}_{k-1}$ as the set of generated parent nodes. The model takes the new set $\mathcal{N}_{k-1} = \mathcal{S}_t^{\text{data}} \cup \mathcal{S}_t^{\text{prev}} \cup \mathcal{S}_t^{\text{aux}} \cup \mathcal{P}_{k-1}$ as the set of all potential choices of child nodes to build the k-th parent node c_k^*. Then we set $\mathcal{P}_k = \mathcal{P}_{k-1} \cup \{c_k^*\}$ and step to the $(k+1)$-th iteration. We further need a hyper-parameter $\bar{N}$ corresponding to the number of iterations of the tree construction steps. The practical algorithm for constructing the computational tree for the t-th hidden cell of RRNN is exhibited in Algorithm 1. It is worth mentioning that the number of iterations $\bar{N}$ in Algorithm 1 might be different from the number of nodes N in the predicted tree. A vector might be chosen several times to serve as a child node in Algorithm 1. In this case, the number of nodes N in the predicted tree is larger than the number of iterations $\bar{N}$.

Algorithm 1 Construction of computational tree for t-th hidden cell

Input: $\mathcal{L}, \mathcal{R}, \mathcal{B}, \mathcal{U}, \mathcal{O}, \alpha, \mathcal{S}_t^{data}, \mathcal{S}_t^{prev}, \mathcal{S}_t^{aux}, \bar{N}$
Output:
  h_t: The hidden state of t-th hidden cell
  T_t^{pred}: Binary computational tree corresponds to the t-th hidden cell

$\mathcal{P}_0 \leftarrow \emptyset$
for $k = 1$ to $\bar{N}$ **do**
  $V_t^k \leftarrow \emptyset$
  for $r = 1$ to $|\mathcal{L}|$, all $o \in \mathcal{O}$, and all $u \in \mathcal{U}$ **do**
    for $c_i, c_j \in \mathcal{S}_t^{data} \cup \mathcal{S}_t^{prev} \cup \mathcal{S}_t^{\text{aux}} \cup \mathcal{P}_{k-1}$, $i < j$ **do**
      $V_t^k \leftarrow V_t^k \cup \{u(o(L_r c_i, R_r c_j) + b_r)\}$
    end for
  end for
  $c_k^* \leftarrow \arg\max \left\{\alpha(v; \Theta) : v \in V_t^k, v \notin \mathcal{P}_{k-1}\right\}$
  $\mathcal{P}_k \leftarrow \mathcal{P}_{k-1} \cup \{c_k^*\}$
end for
$h_t \leftarrow c_{\bar{N}}^*$, $T_t^{\text{pred}} \leftarrow$ the tree rooted at $c_{\bar{N}}^*$
Return h_t and T_t^{pred}

3.3 Loss Function

We discuss the definition of the tree distance (TD) and the scoring margin m in this section.

Score Margin To give scoring more partitioning power, we incentivize it to leave a significant margin between the score of the highest-scoring vector and the second-highest vector for each node. Recall the definition of C_k and c_k^* from Section 3.1. We further define c_k^{**} to be the vector with the second highest score among the vectors in C_k. In Algorithm 1 the analogous to C_k is V_t^k. The scoring margin function is thereby defined as $m(\mathcal{N}_k) = -\frac{1}{M}\min\{M, \alpha(c_k^*; \Theta) - \alpha(c_k^{**}; \Theta)\}$, where M is a hyper-parameter. Intuitively, the margin function incentivizes scoring to increase the gap between the scores of the highest and second-highest vectors to at least M. We divide by M so that the overall scale of this loss term is not affected by the choice of M.

Tree Distance For convenience, in the discussion of this part, we use T^{pd} and T^{tgt} to denote the predicted tree and the target (ground truth) tree, respectively. For any binary tree $\bar{T}$, we use $\text{Int}(\bar{T})$ to denote all internal (non-leaf) nodes of $\bar{T}$. We use $\mathcal{I}(\bar{T})$ to denote the labeling of $\text{Int}(\bar{T})$ such that the root node of $\bar{T}$ has index 1, and if a node has index i, then its left and right child nodes have index $2i$ and $2i+1$, respectively. For a node $n \in \text{Int}(\bar{T})$, we use $\text{Subtree}(\bar{T}, n)$ to denote the subtree of $\bar{T}$ rooted at node n. In addition, we use $n_{i,\bar{T}}$ and $v_{i,\bar{T}}$ to denote the node and the corresponding vector with index i in tree $\bar{T}$, respectively.

Given two binary trees T_1 and T_2, we define

$$\text{VD}(T_1, T_2) = \sum_{i \in \mathcal{I}(T_1) \cap \mathcal{I}(T_2)} \|v_{i,T_1} - v_{i,T_2}\|^2 + \sum_{i \in \mathcal{I}(T_1) \setminus \mathcal{I}(T_2)} \|v_{i,T_1}\|^2 + \sum_{i \in \mathcal{I}(T_2) \setminus \mathcal{I}(T_1)} \|v_{i,T_2}\|^2$$

to be the vector differences (VD) of these two trees. The tree distance between T^{pd} and T^{tgt} is the sum over all minimum VD values between a sub-tree of T^{pd} and all sub-trees of T^{tgt}:

$$\text{TD}(T^{\text{pd}}, T^{\text{tgt}}) = \sum_{n_1 \in V(T^{\text{pd}})} \min_{n_2 \in V(T^{\text{tgt}})} \left\{\text{VD}\big(\text{Subtree}(T^{\text{pd}}, n_1), \text{Subtree}(T^{\text{tgt}}, n_2)\big)\right\}.$$

This expression matches each subtree in T^{pd} with the closet subtree in T^{tgt} with respect to VD, and therefore the TD measures the difference of vectors on all of the nodes of the two trees.

4 Properties of RRNN and Gradient Control

In this section, we state some properties of the RRNN model and show how to avoid gradient exploding and vanishing during training of RRNN. We give theorems in this section and defer the proofs to the appendix.

4.1 Expressibility of RRNN

We argue that if we carefully choose sets $\mathcal{L}, \mathcal{R}, \mathcal{B}, \mathcal{U}, \mathcal{O}, \mathcal{S}_t^{data}, \mathcal{S}_t^{prev}, \mathcal{S}_t^{aux}$, the quantity N, and the scoring function α, then Algorithm 1 can replicate the GRU and LSTM equations. We give the formal statements in this section and defer the choice of the sets and the proof to Appendix C.

Theorem 1. *There exists a scoring function α such that Algorithm 1 generates GRU equations (1) – (4) with an appropriate choice of $\mathcal{L}, \mathcal{R}, \mathcal{B}, \mathcal{U}, \mathcal{O}, \mathcal{S}_t^{data}, \mathcal{S}_t^{prev}$, and $\mathcal{S}_t^{aux}$.*

Theorem 2. *There exists a scoring function α such that Algorithm 1 (applied twice) generates the LSTM equations with an appropriate choice of $\mathcal{L}, \mathcal{R}, \mathcal{B}, \mathcal{U}, \mathcal{O}, \mathcal{S}_t^{data}, \mathcal{S}_t^{prev}$, and $\mathcal{S}_t^{aux}$.*

4.2 Controlling Gradient

As introduced in [2], the exploding gradient problem refers to the large increase in the norm of the gradient during training. This is due to the fact that the gradient of long-term dependencies grows exponentially quicker than for short-term dependencies. The vanishing gradient problem, on the other hand, refers to the behavior that the gradients of long-term dependencies go to zero exponentially. [20] introduce a sufficient condition of vanishing gradient and a necessary condition of exploding gradient for a simple RNN. In this section, we extend their results to a more general case – we provide these two conditions for our RRNN model. We note that our result as a special case applies to RecNN where such conditions have not yet been established.

We consider the case where only one hidden state h_t is returned by the t-th hidden cell of the RRNN model. The loss function (6) can be written as $L(\Phi) = \sum_{t=1}^{T} \mathcal{E}_t$ where each $\mathcal{E}_t$ is a function of all parameters in Φ. For $1 \leq t \leq T$, the gradient of $\mathcal{E}_t$ with respect to $\phi \in \Phi$ comes from t cells, namely $\frac{\partial \mathcal{E}_t}{\partial \phi} = \sum_{t'=1}^{t} \frac{\partial \mathcal{E}_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial^+ h_{t'}}{\partial \phi}$, where $\frac{\partial^+ h_{t'}}{\partial \phi}$ refers to the direct gradient of $h_{t'}$ with respect to ϕ directly appearing within the t'-th hidden cell. If ϕ is a matrix, then we mean $\frac{\partial^+ h_{t'}}{\partial \phi} = \frac{\partial^+ h_{t'}}{\partial \mathrm{vec}(\phi)}$, where $\mathrm{vec}(\phi)$ is an appropriate matrix vectorization. The exploding (vanishing) gradient problem is defined by $\left\| \frac{\partial \mathcal{E}_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial^+ h_{t'}}{\partial \phi} \right\|$ going to $+\infty$ (0) exponentially fast as t goes to $+\infty$ and t' is fixed as a constant. For simplicity, we consider the case where $t = T$ and $t' = 1$.

We state a simplified version of the theorems here and defer the full version to Appendix D. We argue that most of the time these conditions are met in practice and we elaborate them one by one in Appendix D.2.

Theorem 3 (Sufficient condition of gradient vanishing)**.** *Under certain conditions given in Theorem 7, we have $\left\| \frac{\partial \mathcal{E}_T}{\partial h_T} \frac{\partial h_T}{\partial h_1} \frac{\partial^+ h_1}{\partial \phi} \right\| \to 0$ as $T \to +\infty$, i.e., the vanishing gradient problem occurs.*

Theorem 4 (Necessary condition of gradient exploding)**.** *If we observe the vanishing gradient problem, then at least one of the conditions listed in Theorem 8 holds.*

5 Experimental Results

In this section, we present numerical results by comparing our algorithm with a GRU baseline model. The experiments are conducted on three datasets, and the source code is available at `http://after_accepted`.

We test two versions of the RRNN algorithm. The first one is the full algorithm we presented in Algorithm 1. The second one, which we call it RRNN-GRU, is a simplified version of the RRNN model where we limit the tree structure to be exactly the same as GRU. This model has a limited tree search space and the only dynamic component is the choice of the tuple (L_i, R_i, b_i) to use on each pair of parent-child nodes, so the positioning of weights in the cell is flexible. Therefore, RRNN-GRU is still time-variant and data-dependent. In addition, we alternate between training the L, R, b

Table 1: Performance of models on three datasets

	Wiki-5k (BPC)		Wiki-10k (BPC)	
	Val	Test	Val	Test
RRNN	**2.58 (-5.5%)**	**2.63 (-1.9%)**	–	–
RRNN-GRU	–	–	**2.43 (-5.8%)**	**2.42 (-5.8%)**
GRU	2.73	2.68	2.58	2.57
	SST (Accuracy)		PTB (Perplexity)	
	Val	Test	Val	Test
RRNN-GRU	**65.1% (-0.8%)**	**68.7% (-5.2%)**	281	239
GRU	64.6%	65.3 %	**247 (-12.1%)**	239

parameters and training the scoring neural network α consisting of a 2-layer fully connected neural network, while continuously training the output layer. The frequency (in epochs) that we switch training phases is set as a hyperparameter of the RRNN-GRU model. Due to the model architecture, training can sometimes be unstable with exploding gradients which we clip. The baseline model is the single layer GRU which has 100-dimensional hidden states.

For both 100-dimensonal character and word embeddings, we used the pre-trained embedding vectors from GloVe[1]. The Adam optimizer is used for all experiments and random initial weights are selected. A random search on hyperparameters is used for all RRNN-GRU models and GRU models. We train the model parameters on the training set and select the optimal parameters and hyperparameters based on the performance measure on the validation set. Then we use this set of hyperparameters and the optimized model parameters to predict on the test set. We test the RRNN model only on the Wikipedia dataset. We report the performance on both validation and test sets for all datasets in Table 1 and list the optimal hyperparamers in Appendix E.2. Further details about the implementations are given in Appendix E.1.

5.1 Datasets and Settings

The *Wikipedia* task is to predict the next character on text drawn from the Hutter prize Wikipedia dataset[2] [13]. We remove all numbers, punctuation, XML tags, and markup characters so that 26 English characters and space are left in the raw text. Performance is measured using BPC (the smaller the better). For RRNN-GRU, we randomly select 10,000 20-character sequences for the training set, along with 1,000 sequences for validation and 2,000 for testing, such that no sequences overlap. For RRNN, the training set has 5,000 sequences while the validation and test sets remain of the same size.

The Stanford Sentiment Treebank (SST) dataset[3] [25] is a sentiment analysis task involving classifying one-sentence movie reviews as positive, negative, or neutral. We obtain the dataset from the `torchtext` package and use the full 8,544-sample training set, along with a randomly-chosen 1,000 samples for validation and 2,000 for testing. Since the training data has variable length, we prepend each sample with zeros to make each sample be the same length. The performance is measured in the accuracy of correctly predicting sentiments (the higher the better).

We also perform word-level language modeling using the *Penn Treebank* (PTB) dataset[4] [18], a corpus containing articles from the Wall Street Journal. We obtain this dataset from the `torchtext` package and randomly select a 10,000 sample subset of 20 words each, along with 1,000 samples for validation and 2,000 for testing. We predict over all 10,001 unique words in our subset without eliminating uncommon words. The performance is measured in perplexity (the smaller the better).

[1] https://nlp.stanford.edu/projects/glove/
[2] https://cs.fit.edu/~mmahoney/compression/textdata.html
[3] https://nlp.stanford.edu/sentiment/treebank.html
[4] https://catalog.ldc.upenn.edu/LDC99T42

5.2 Discussion

From Table 1 it is clear that RRNN-GRU outperforms GRU by 5.8% on the Wikipedia dataset while RRNN improves the results of GRU by 5.5% on validation set and 1.9% on test set with a simple set of hyperparameters. On SST, RRNN-GRU also beats GRU by 0.8% and 5.2% on validation and test sets, respectively. These experiments show that the data-dependent structures do help improve the prediction power of the model and achieve better performance. Meanwhile, RRNN-GRU matches the performance of GRU on PTB. Its performance on the test set of PTB can be improved by a more dedicated hyperparameter search.

One interesting observation of the full RRNN model is the evolution of the predicted tree structures. Figure 3 of Appendix A shows the common tree structures we find at the beginning epochs while Figure 4 of Appendix A shows the common tree structures at later epochs (near the point where optimal performance is achieved on the validation set). The tree structures tend to be balanced in the beginning epochs since the structure of the ground-truth tree plays a significant role. In later epochs, the output layer dominates the predicting ability and therefore the model tends to feed simple h_t to the output layer.

Another interesting observation lies in the dynamics of the RRNN-GRU model. Let us denote $\mathcal{I}_{e,i,t,j}$ to be the index of parameter tuple (L, R, b) that the j-th internal node of i-th sample on the t-th time step in e-th epoch, and we further set $N_e \triangleq \sum_{i,t,j} \mathbb{1}\{\mathcal{I}_{e,i,t,j} \neq \mathcal{I}_{e-1,i,t,j}\}$ to measure the number of changes in the choice of parameter tuples between $(e-1)$-th epoch and e-th epoch. Then we should expect the quantity N_e to be decreasing as e increases since the model is expected to become more stable as the training goes on and the choice of indices of parameter tuples should also become more stable. Figure 2 in Appendix A shows the plot of N_e vs epochs which supports our hypothesis.

Acknowledgments

The authors would also like to acknowledge and thank Intel for providing access to Intel's Computing environment.

References

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[3] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations (ICLR)*, 2017.

[4] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1251–1258, 2017.

[5] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Conference on Neural Information Processing Systems (NIPS) Workshop on Deep Learning*, 2014.

[6] Wim De Mulder, Steven Bethard, and Marie-Francine Moens. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, 30(1):61–98, 2015.

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[8] Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN)*, pages 347–352, 1996.

[9] Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

[10] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610, 2005.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[12] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[13] Marcus Hutter. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media, 2004.

[14] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

[15] Xiangang Li and Xihong Wu. Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4520–4524. IEEE, 2015.

[16] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[17] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations (ICLR)*, 2019.

[18] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[19] W. Thomas Miller, Paul J. Werbos, and Richard S. Sutton. *Neural networks for control*. MIT press, 1995.

[20] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning (ICML)*, pages 1310–1318, 2013.

[21] Hieu Pham, Melody Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning (ICML)*, pages 4092–4101, 2018.

[22] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH)*, pages 338–342, 2014.

[23] Martin Schrimpf, Stephen Merity, James Bradbury, and Richard Socher. A flexible approach to automated RNN architecture generation. *arXiv preprint arXiv:1712.07316*, 2017.

[24] Richard Socher, Cliff C. Lin, Christopher D. Manning, and Andrew Ng. Parsing natural scenes and natural language with recursive neural networks. In *International Conference on Machine Learning (ICML)*, pages 129–136, 2011.

[25] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1631–1642. Association for Computational Linguistics, 2013.

[26] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *International Conference on Learning Representations (ICLR)*, 2016.

[27] Lida Zhang and Diego Klabjan. Layer flexible adaptive computational time for recurrent neural networks. *arXiv preprint arXiv:1812.02335*, 2018.

[28] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

[29] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8697–8710, 2018.

Appendices

A Figures

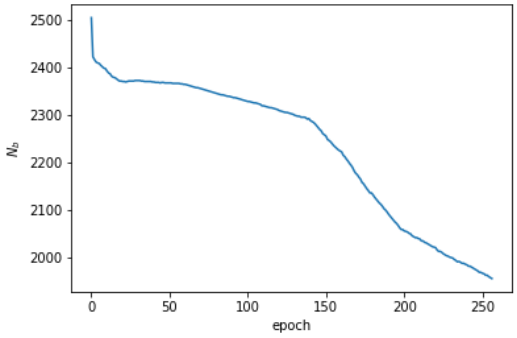
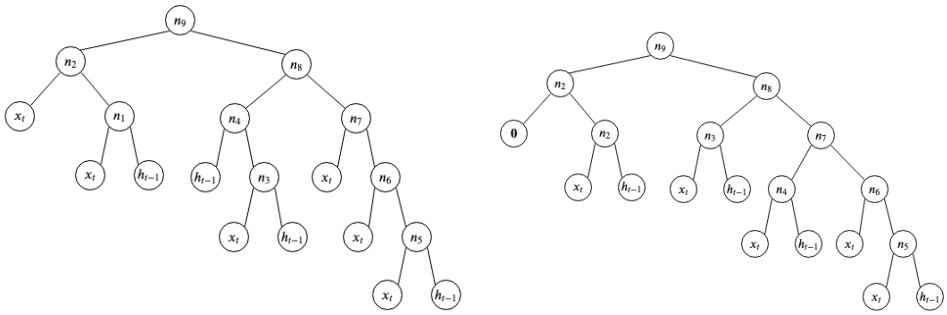
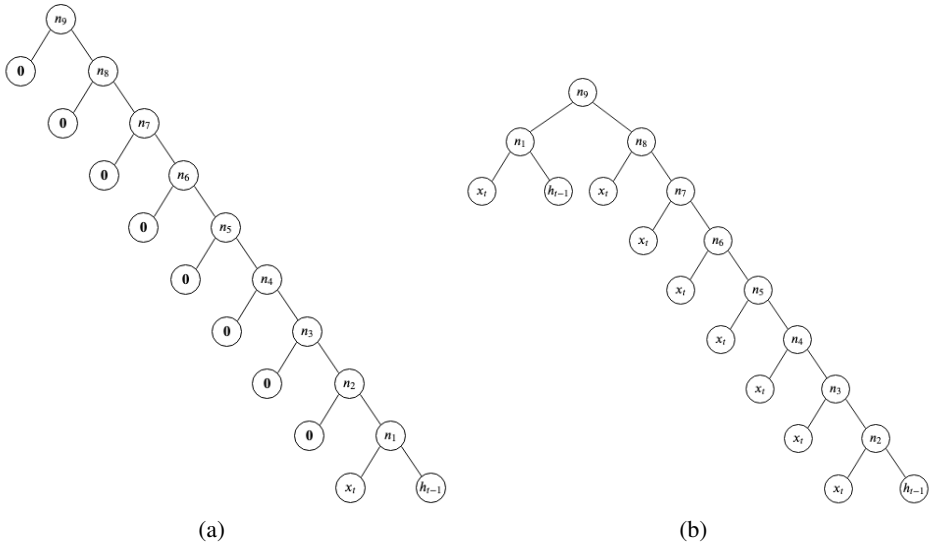


Figure 2: The number of changes in the indices of parameter tuples vs epochs



(a) (b)

Figure 3: Example of tree structures at early stage of training.



(a) (b)

Figure 4: Example of tree structures near the optimal stage of training.

B Extensions of RRNN model

The exposition so far handles the case where only one state vector transfers between hidden cells of the RRNN model, and it can capture the structure of GRU. However, LSTM, for example, has two state vectors h_t and c_t to transfer between cells. In this section we extend the RRNN model to be compatible with transferring multiple state vectors. Suppose that a total of M vectors, $h_{t-1,1}, \ldots, h_{t-1,M}$, are the output of the $(t-1)$-th hidden cell. The transition equations and cell output are thereby

$$
\begin{aligned}
&(T_{t,i}^{\text{pred}}, h_{t,i}) = f^i(\mathcal{N}_{0,i}^t) \quad i = 1, 2, \ldots, M, \\
&q_t = g(x_t, h_{t,M}; \Gamma),
\end{aligned}
$$

where each $f^i \triangleq f_N^i \circ \cdots \circ f_1^i$ has the same definition as the function f defined in (5), and $\mathcal{N}_{0,i}^t$ is the multiset consisting of multiple copies of x_t, $h_{t-1,j}, j = 1, \ldots, M$, $h_{t,j}, j = 1, \ldots, i-1$, and possible other constant vectors. In practice, we use Algorithm 1 M times to build functions $f^i, i = 1, 2, \ldots, M$.

The loss function is redefined as

$$
\begin{aligned}
L(\Phi) = \mathbb{E}_{(X,Y)} \Bigg[\sum_{t=1}^{T} \Bigg\{ \lambda_1 l(y_t, q_t) + \lambda_2 \sum_{i=1}^{M} \min_{\bar{T} \in \text{Iso}(T_{t,i}^{\text{target}})} \text{TD}(\bar{T}, T_{t,i}^{\text{pred}}) + \lambda_3 \sum_{i=1}^{M} \sum_{k=0}^{N-1} m(\mathcal{N}_{k,i}^t) \Bigg\} \Bigg] \\
+ \lambda_4 \sum_{\phi \in \Phi} \|\phi\|^2,
\end{aligned}
$$

where $\mathcal{N}_{k,i}^t = f_k^i \circ \cdots \circ f_1^i(\mathcal{N}_{0,i}^t), k = 1, 2, \ldots, N-1, i = 1, 2, \ldots, M$, and $T_{t,i}^{\text{target}}$ is the ground truth binary tree.

As an example, we show how to transfer two state vectors c_t and h_t between hidden cells of RRNN and mimic the structure of LSTM. To adhere with notation from prior works, we use c_t and h_t to replace $h_{t,1}$ and $h_{t,2}$ in the above general definition. The transition equations and cell output are therefore

$$
\begin{aligned}
&(T_{t,1}^{\text{pred}}, c_t) = f^1(\mathcal{N}_{0,1}^t(x_t, c_{t-1}, h_{t-1})), \\
&(T_{t,2}^{\text{pred}}, h_t) = f^2(\mathcal{N}_{0,2}^t(x_t, c_{t-1}, h_{t-1}, c_t)), \\
&q_t = g(x_t, h_t; \Gamma),
\end{aligned}
$$

where $\mathcal{N}_{0,1}^t(x_t, c_{t-1}, h_{t-1})$ consists of several copies of x_t, h_{t-1}, c_{t-1} and possible other constant vectors, and $\mathcal{N}_{0,2}^t(x_t, c_{t-1}, h_{t-1}, c_t)$ consists of several copies of $x_t, c_{t-1}, h_{t-1}, c_t$ and possible other constant vectors.

C Expressibility of RRNN

We extend the contect of Section 5 here. We first show that for a given set of vectors, there always exists a scoring function that can rank the scores of these vectors by any order we want. Formally, we have the following lemma.

Lemma 1. *Given n vectors $v_1, \ldots, v_n \in \mathbb{R}^p$, there exists a function α with a set of parameters Θ such that $\alpha(v_1; \Theta) > \cdots > \alpha(v_n; \Theta)$.*

We next show that if we carefully choose sets $\mathcal{L}, \mathcal{R}, \mathcal{B}, \mathcal{U}, \mathcal{O}, \mathcal{S}_t^{data}, \mathcal{S}_t^{prev}, \mathcal{S}_t^{aux}$, the quantity N, and the scoring function α, then Algorithm 1 can replicate the GRU and LSTM equations.

To replicate GRU, we should have $N = 8$, $\mathcal{S}_t^{data} = \{x_t\}$, $\mathcal{S}_t^{prev} = \{h_{t-1}\}$, $\mathcal{S}_t^{aux} = \{\mathbf{0}\}$, where $\mathbf{0}$ is the zero vector. We further set

- $\mathcal{L} = \{L_1, L_2, L_3, L_4\}$, where $L_1 = W_r, L_2 = W_z, L_3 = W_h$, and $L_4 = I$,
- $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$, where $R_1 = W_r', R_2 = W_z', R_3 = W_h'$, and $R_4 = I$,
- $\mathcal{B} = \{b_1, b_2, b_3, b_4\}$, where $b_1 = b_r, b_2 = b_z, b_3 = b_h$, and $b_4 = \mathbf{0}$,

- $\mathcal{U} = \{\sigma(\cdot), \tanh(\cdot), \mathbf{1} - \cdot, \mathrm{id}(\cdot)\}$, where $\mathbf{1}$ stands for the all-ones vector and id stands for the identity mapping,
- $\mathcal{O} = \{+, \odot\}$, where $\odot$ is the entry-wise multiplication.

Theorem 1 therefore becomes the following

Theorem 5. *There exists a scoring function α such that Algorithm 1 generates GRU equations (1) – (4) for the choice of $\mathcal{L}, \mathcal{R}, \mathcal{B}, \mathcal{U}, \mathcal{O}, \mathcal{S}_t^{data}, \mathcal{S}_t^{prev}$, and $\mathcal{S}_t^{aux}$ specified above.*

For LSTM, note that there are two state vectors h_t and c_t. Therefore, to replicate LSTM (see equations (7) – (11) below), we run Algorithm 1 twice. In the first run, we should have $N = 7$, $\mathcal{S}_t^{data} = \{x_t\}$, $\mathcal{S}_t^{prev} = \{h_{t-1}, c_{t-1}\}$, $\mathcal{S}_t^{aux} = \{\mathbf{0}\}$. We further set

- $\mathcal{L} = \{L_6\}$, where $L_6 = W_f, L_2 = W_i, L_3 = W_o, L_4 = W_c$, and $L_5 = I$,
- $\mathcal{R} = \{R_1, R_2, R_3, R_4, R_5\}$, where $R_1 = W_f', R_2 = W_i', R_3 = W_o', R_4 = W_c'$, and $R_5 = I$,
- $\mathcal{B} = \{b_1, b_2, b_3, b_4, b_5\}$, where $b_1 = b_f, b_2 = b_i, b_3 = b_o, b_4 = b_c$, and $b_5 = \mathbf{0}$,
- $\mathcal{U} = \{\sigma(\cdot), \tanh(\cdot), \mathrm{id}(\cdot)\}$,
- $\mathcal{O} = \{+, \odot\}$.

In the second run, we should have $N = 2$, $\mathcal{S}_t^{data} = \{x_t\}$, $\mathcal{S}_t^{prev} = \{h_{t-1}, c_{t-1}, c_t\}$, $\mathcal{S}_t^{aux} = \{\mathbf{0}\}$. We further set $\mathcal{L} = \{L_6\}, \mathcal{R} = \{R_6\}$, where $L_6 = R_6 = I$, $\mathcal{B} = \{b_6\}$, where $b_6 = \mathbf{0}$, $\mathcal{U} = \{\tanh(\cdot), \mathrm{id}(\cdot)\}$, and $\mathcal{B} = \{+, \odot\}$. Theorem 2 therefore becomes the following

Theorem 6. *There exists a scoring function α such that Algorithm 1 (applied twice) generates the following LSTM equations*

$$f_t = \sigma\left(W_f x_t + W_f' h_{t-1} + b_f\right) \tag{7}$$
$$i_t = \sigma\left(W_i x_t + W_i' h_{t-1} + b_i\right) \tag{8}$$
$$o_t = \sigma\left(W_o x_t + W_o' h_{t-1} + b_o\right) \tag{9}$$
$$c_t = c_{t-1} \odot f_t + i_t \odot \tanh\left(W_c x_t + W_c' h_{t-1} + b_c\right) \tag{10}$$
$$h_t = o_t \odot \tanh(c_t) \tag{11}$$

for the choice of $\mathcal{L}, \mathcal{R}, \mathcal{B}, \mathcal{U}, \mathcal{O}, \mathcal{S}_t^{data}, \mathcal{S}_t^{prev}$, and $\mathcal{S}_t^{aux}$ specified above.

C.1 Proof of Lemma 1

Consider function

$$\alpha(v; \Theta) = \sum_{k=1}^{n} k \exp\left(-\frac{\|v - v_k\|^2}{2\sigma_0^2}\right)$$

with $\Theta = \{\sigma_0\}$, and σ_0 is a large enough constant such that $\sigma_0^2 \geq \frac{\Delta}{\log(n^2) - \log(n^2-1)}$, and $\Delta = \frac{1}{2}\min_{j \neq k} \|v_j - v_k\|^2$.

For $0 \leq i \leq n-1$, since $-\frac{\|v_{i+1} - v_k\|^2}{2\sigma_0^2} \leq 0$, we have

$$\alpha(v_{i+1}; \Theta) = \sum_{k=1}^{n} k \exp\left(-\frac{\|v_{i+1} - v_k\|^2}{2\sigma_0^2}\right) \leq \sum_{\substack{1 \leq k \leq n \\ k \neq i+1}} k = \frac{n(n+1)}{2} - (i+1)\,.$$

On the other hand, for a fixed $1 \leq i \leq n$ and all $1 \leq k \leq n$, we have $\exp\left(-\frac{\|v_i - v_k\|^2}{2\sigma_0^2}\right) \geq \exp\left(-\frac{\Delta}{\sigma_0^2}\right) \geq \exp\left(\log(n^2-1) - \log(n^2)\right) = 1 - \frac{1}{n^2} \geq 1 - \frac{1}{nk}$, and therefore

$$\alpha(v_i; \Theta) = \sum_{k=1}^{n} k \exp\left(-\frac{\|v_i - v_k\|^2}{2\sigma_0^2}\right) \geq \sum_{\substack{1 \leq k \leq n \\ k \neq i}} k\left(1 - \frac{1}{nk}\right)$$
$$= \frac{n(n+1)}{2} - i - \frac{n-1}{n} > \frac{n(n+1)}{2} - (i+1)\,.$$

In conclusion, for $1 \leq i \leq n-1$, we have

$$\alpha(v_{i+1};\Theta) \leq \frac{n(n+1)}{2} - (i+1) < \alpha(v_i;\Theta)\,,$$

and thus $\alpha(v_1;\Theta) > \cdots > \alpha(v_n;\Theta)$.

C.2 Proof of Theorem 5 and 6

We start with the proof of Theorem 5. In Algorithm 1, we set $N = 8$, $\mathcal{S}_t^{data} = \{x_t\}$, $\mathcal{S}_t^{prev} = \{h_{t-1}\}$, $\mathcal{S}_t^{aux} = \{\mathbf{0}\}$, where $\mathbf{0}$ is the zero vector. In the following, "the algorithm" refers to Algorithm 1. The scoring function α has a set of parameters Θ and is capable of sorting the scores of different vectors. We show the existence of this function at the end of the proof by relying on Lemma 1.

We start with $\mathcal{P}_0 = \emptyset$. For $k = 1$, the algorithm generates the vector set V_t^1 and one of its elements is $r_t \triangleq \sigma\left(L_1 x_t + R_1 h_{t-1} + b_1\right) = \sigma\left(W_r x_t + W_r' h_{t-1} + b_r\right)$. The scoring function α guarantees that $\alpha(r_t;\Theta) > \alpha(v;\Theta), \forall v \in V_t^1, v \neq r_t$. Therefore, we have $c_1^* = r_t$ and $\mathcal{P}_1 = \{r_t\}$. Similarly, the algorithm finds z_t to be the vector with the highest score in the set $V_t^2 \setminus \mathcal{P}_1$. We have $c_2^* = z_t$ and $\mathcal{P}_2 = \{r_t, z_t\}$.

For $k = 3$, the algorithm generates the vector set V_t^3 and one of its elements is $\widetilde{r}_t \triangleq \mathrm{id}[(L_4 h_{t-1}) \odot (R_4 r_t) + b_4] = r_t \odot h_{t-1}$. The scoring function α guarantees that $\alpha(\widetilde{r}_t;\Theta) > \alpha(v;\Theta), \forall v \in V_t^3 \setminus \mathcal{P}_2, v \neq \widetilde{r}_t$. Therefore, we have $c_3^* = \widetilde{r}_t$ and $\mathcal{P}_3 = \{r_t, z_t, \widetilde{r}_t\}$.

For $k = 4$, the algorithm generates the vector set V_t^4 and one of its elements is $\mathbf{1} - z_t = \mathbf{1} - (L_4\mathbf{0} + R_4 z_t + b_4)$. The scoring function α guarantees that $\alpha(\mathbf{1}-z_t;\Theta) > \alpha(v;\Theta), \forall v \in V_t^4 \setminus \mathcal{P}_3, v \neq \mathbf{1}-z_t$. Therefore, we have $c_4^* = \mathbf{1} - z_t$ and $\mathcal{P}_4 = \{r_t, z_t, \widetilde{r}_t, \mathbf{1} - z_t\}$.

For $k = 5$, the algorithm generates the vector set V_t^5 and one of its elements is $\widetilde{h}_t \triangleq \tanh(L_3 x_t + R_3 \widetilde{r}_t + b_3) = \tanh(W_h x_t + W_h'(r_t \odot h_{t-1}) + b_h)$. The scoring function α guarantees that $\alpha(\widetilde{h}_t;\Theta) > \alpha(v;\Theta), \forall v \in V_t^5 \setminus \mathcal{P}_4, v \neq \widetilde{h}_t$. Therefore, we have $c_5^* = \widetilde{h}_t$ and $\mathcal{P}_5 = \{r_t, z_t, \widetilde{r}_t, \mathbf{1} - z_t, \widetilde{h}_t\}$.

For $k = 6$, the algorithm generates the vector set V_t^6 and one of its elements is $z_t \odot h_{t-1} = \mathrm{id}[(L_4 h_{t-1}) \odot (R_4 z_t) + b_4]$. The scoring function α guarantees that $\alpha(z_t \odot h_{t-1};\Theta) > \alpha(v;\Theta), \forall v \in V_t^6 \setminus \mathcal{P}_5, v \neq z_t \odot h_{t-1}$. Therefore, we have $c_6^* = z_t \odot h_{t-1}$ and $\mathcal{P}_6 = \{r_t, z_t, \widetilde{r}_t, \mathbf{1} - z_t, \widetilde{h}_t, z_t \odot h_{t-1}\}$. Similarly, the algorithm finds $(\mathbf{1} - z_t) \odot \widetilde{h}_t$ to be the vector with the highest score in the set $V_t^7 \setminus \mathcal{P}_6$. Thus we have $c_7^* = (\mathbf{1} - z_t) \odot \widetilde{h}_t$ and $\mathcal{P}_7 = \{r_t, z_t, \widetilde{r}_t, \mathbf{1} - z_t, \widetilde{h}_t, z_t \odot h_{t-1}, (\mathbf{1} - z_t) \odot \widetilde{h}_t\}$

Finally, for $k = 8$, the algorithm generates the vector set V_t^8 and one of its elements is $h_t \triangleq \mathrm{id}[(L_4(z_t \odot h_{t-1})) + (R_4((1 - z_t) \odot \widetilde{h}_t)) + b_4] = z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h}_t$. The scoring function α guarantees that $\alpha(h_t;\Theta) > \alpha(v;\Theta), \forall v \in V_t^8 \setminus \mathcal{P}_7, v \neq h_t$. Therefore, we have $c_8^* = h_t$ and $\mathcal{P}_8 = \{r_t, z_t, \widetilde{r}_t, \mathbf{1} - z_t, \widetilde{h}_t, z_t \odot h_{t-1}, (\mathbf{1} - z_t) \odot \widetilde{h}_t, h_t\}$.

It remains to specify the scoring function α. Note that α should satisfy that $\alpha(c_i^*;\Theta) > \alpha(v;\Theta), \forall v \in V_t^i \setminus \mathcal{P}_{i-1}, v \neq c_i^*$ for $1 \leq i \leq 8$. Since each set V_t^i contains a finite number of vectors, Lemma 1 guarantees that such scoring function α exists. Therefore, the algorithm returns vector h_t and the binary tree rooted at h_t as the output, and thus it replicates the GRU equations (1) – (4). □

The proof of Theorem 6 is almost the same as the proof above. The only difference is that in the first run of Algorithm 1, we generate equations (7) – (10), while in the second run of Algorithm 1, we generate equation (11).

D Gradient Control

In this section, we first give the full statements of Theorem 3 and 4. Then we give proofs for these two theorems and discuss about them.

Theorem 7 (Sufficient condition of gradient vanishing). *Let $\mathcal{V}_t$ to be the set of vectors on the nodes of predicted tree T_t^{pred}. Assume that there exist constants C_1, C_2, C_3, C_4 such that for all $1 \leq t \leq T$,*

$$\|L\| \leq C_1, \forall L \in \mathcal{L}, \|R\| \leq C_1, \forall R \in \mathcal{R}, \tag{12}$$

$$\|u'\|_\infty \leq C_2, \forall u \in \mathcal{U}, \tag{13}$$

$$\left\|\frac{\partial o(L_i v_1, R_i v_2)}{\partial (L_i v_1)}\right\| \leq C_3, \left\|\frac{\partial o(L_i v_1, R_i v_2)}{\partial (R_i v_2)}\right\| \leq C_3, 1 \leq i \leq n_l, \forall v_1, v_2 \in \mathcal{V}_t, v_1 \neq v_2, \forall o \in \mathcal{O}, \tag{14}$$

$$\left\|\frac{\partial \mathcal{E}_t}{\partial h_t}\right\| \leq C_4, t = 0, 1, \ldots, \tag{15}$$

$$C_1 C_2 C_3 < \frac{1}{2}. \tag{16}$$

Under conditions (12) – (16), we have $\left\|\frac{\partial \mathcal{E}_T}{\partial h_T}\frac{\partial h_T}{\partial h_1}\frac{\partial^+ h_1}{\partial \phi}\right\| \to 0$ as $T \to +\infty$, i.e., the vanishing gradient problem occurs.

Theorem 8 (Necessary condition of gradient exploding, restated). *Let $l_{\min} \triangleq \lfloor \log_2(N+1) \rfloor + 1$ be the minimum possible depth of all full binary trees $T_t^{\text{pred}}, 1 \leq t \leq T$. If the exploding gradient problem occurs, then at least one of the following conditions hold:*

- *there exists an activation function $u \in \mathcal{U}$ such that $\|u'\| \geq (N+1)^{-\frac{1}{3l_{\min}}}$,*
- *there exists a parameter matrix $P \in \mathcal{L} \cup \mathcal{R}$ such that $\|P\| \geq (N+1)^{-\frac{1}{3l_{\min}}}$,*
- *for infinite many t, there exists a pair of parent-child nodes (v, v_1) in the tree T_t^{pred} such that $\left\|\frac{\partial o(L_i v_1, R_i v_2)}{\partial (L_i v_1)}\right\| \geq (N+1)^{-\frac{1}{3l_{\min}}}$, where $v = u(o(L_i v_1, R_i v_2) + b_i)$,*
- *for infinite many t, there exists a pair of parent-child nodes (v, v_2) in the tree T_t^{pred} such that $\left\|\frac{\partial o(L_i v_1, R_i v_2)}{\partial (R_i v_2)}\right\| \geq (N+1)^{-\frac{1}{3l_{\min}}}$, where $v = u(o(L_i v_1, R_i v_2) + b_i)$.*

D.1 Proof of Theorem 7

Recall that Algorithm 1 builds a full binary tree T_t^{pred} with N internal nodes and $N+1$ leaf nodes for the t-th hidden cell of the RRNN model. We use $n_{1,t}^{\text{int}}, \ldots, n_{N,t}^{\text{int}}$ and $n_{1,t}^{\text{leaf}}, \ldots, n_{N+1,t}^{\text{leaf}}$ to denote the internal nodes and leaf nodes of the tree T_t^{pred}, respectively. We denote $\mathcal{V}_t = \{v_{n_{1,t}^{\text{int}}}, \ldots, v_{n_{N,t}^{\text{int}}}, v_{n_{1,t}^{\text{leaf}}}, \ldots, v_{n_{N+1,t}^{\text{leaf}}}\}$ to be the set of vectors on the predicted tree T_t^{pred}. We use $\|A\|$ and $\|v\|_\infty$ to denote the spectral norm of matrix A and the infinity norm of vector v, respectively. We use $\text{diag}\{v\}$ to denote the diagonalization of vector v. For an activation function $u \in \mathcal{U}$, we use u' to denote the derivative of u.

Note that

$$\frac{\partial \mathcal{E}_T}{\partial \phi} = \sum_{t'=1}^{T} \frac{\partial \mathcal{E}_T}{\partial h_T}\frac{\partial h_T}{\partial h_{t'}}\frac{\partial^+ h_{t'}}{\partial \phi} = \sum_{t'=1}^{T} \frac{\partial \mathcal{E}_T}{\partial h_T}\left(\prod_{t'<t\leq T}\frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial^+ h_{t'}}{\partial \phi}.$$

Intuitively, the vanishing gradients problem appears when the norm of $\frac{\partial h_i}{\partial h_{t-1}}$ is smaller than 1. We first provide some lemmas that facilitate proving Theorem 3. For simplicity, we remove the subscript t in $n_{k,t}^{\text{int}}$ and $n_{k,t}^{\text{leaf}}$, since the following derivation applies to all $1 \leq t \leq T$ in the same way.

We define the path starting from the root node n_N^{int} to a leaf node n_k^{leaf} by $P^k = [P_0^k, P_1^k, \ldots, P_{l_k}^k]$, where l_k is the length of this path, $P_0^k = n_k^{\text{leaf}}$, and $P_{l_k}^k = n_N^{\text{int}}$. Lemma 2 gives an upper-bound for the norm of the gradient of a node with respect to one of its child node in the binary tree T_t^{pred}.

Lemma 2. *Under conditions (12) – (15), there exists a constant $C_0 < \frac{1}{2}$ such that $\left\|\frac{\partial P_j^k}{\partial P_{j-1}^k}\right\| \leq C_0$ for all $1 \leq k \leq N+1$ and $1 \leq j \leq l_k$.*

Proof. For simplicity, we write $v = u(o(L_i v_1, R_i v_2) + b_i)$, where i is the index in $\mathcal{L}, \mathcal{R}$ and $\mathcal{B}$, $v = P_j^k$, $v_1 = P_{j-1}^k$, and v_2 is the other child node of P_j^k. The case where $v_2 = P_{j-1}^k$ is similar.

By the chain rule, we have

$$\begin{aligned}\left\|\frac{\partial v}{\partial v_1}\right\| &= \left\|\frac{\partial u(o(L_iv_1, R_iv_2)+b_i)}{\partial[o(L_iv_1, R_iv_2)+b_i]}\frac{\partial[o(L_iv_1, R_iv_2)+b_i]}{\partial(L_iv_1)}\frac{\partial(L_iv_1)}{\partial v_1}\right\| \\ &= \left\|\text{diag}\left\{u'(o(L_iv_1, R_iv_2)+b_i)\right\}\frac{\partial o(L_iv_1, R_iv_2)}{\partial(L_iv_i)}L_i\right\| \\ &\le \|\text{diag}\left\{u'(o(L_iv_1, R_iv_2)+b_i)\right\}\|\left\|\frac{\partial o(L_iv_1, R_iv_2)}{\partial(L_iv_i)}\right\|\|L_i\| \\ &\le C_1C_2C_3,\end{aligned}$$

where the last inequality follows by $\|\text{diag}\{u'\}\| = \|u'\|_\infty \le C_3$ by condition (14) together with conditions (12) and (13). By setting $C_0 = C_1C_2C_3$, the statement holds from condition (16). □

We have for all $1 \le t \le T$,

$$\begin{aligned}\left\|\frac{\partial h_t}{\partial h_{t-1}}\right\| &= \left\|\sum_{k=1}^{N+1}\frac{\partial h_t}{\partial n_k^{\text{leaf}}}\mathbb{1}\left\{n_k^{\text{leaf}} = h_{t-1}\right\}\right\| \le \left\|\sum_{k=1}^{N+1}\frac{\partial h_t}{\partial n_k^{\text{leaf}}}\right\| \le \sum_{k=1}^{N+1}\left\|\frac{\partial h_t}{\partial n_k^{\text{leaf}}}\right\| = \sum_{k=1}^{N+1}\left\|\frac{\partial P_{l_k}^k}{\partial P_0^k}\right\| \\ &= \sum_{k=1}^{N+1}\left\|\prod_{j=1}^{l_k}\frac{\partial P_j^k}{\partial P_{j-1}^k}\right\| \le \sum_{k=1}^{N+1}\prod_{j=1}^{l_k}\left\|\frac{\partial P_j^k}{\partial P_{j-1}^k}\right\| \le \sum_{k=1}^{N+1}C_0^{l_k},\end{aligned} \tag{17}$$

where the last inequality follows by Lemma 2.

Note that $l_1, \ldots, l_{N+1}$ are the lengths of all the paths starting from the root node to the leaf node of a full binary tree. Lemma 3 gives an upper-bound of the sum of exponents of these lengths.

Lemma 3. *Suppose $l_k, 1 \le k \le N+1$ are the lengths of the $N+1$ paths of the full binary tree T_t^{pred}. Then for any $0 < C_0 < \frac{1}{2}$, there exists a constant $\epsilon = \epsilon(C_0), 0 < \epsilon < 1$, such that*

$$\sum_{k=1}^{N+1}C_0^{l_k} \le 1 - \epsilon.$$

Proof. We prove by induction on N that

$$\sum_{k=1}^{N+1}C_0^{l_k} \le C_0^N + \sum_{k=1}^{N}C_0^k. \tag{18}$$

For $N = 1$, the tree T_t^{pred} has exactly one internal node and two leaf nodes, thus there is only one tree structure for T_t^{pred} if we do not consider isomorphisms. We have $\{l_1, l_2, l_3\} = \{2, 2, 1\}$ and $\sum_{k=1}^{N+1}C_0^{l_k} = 2C_0^2 + C_0 = C_0^N + \sum_{k=1}^{N}C_0^k$.

Suppose that equation (18) holds for $N \ge 1$, and we consider the case of $N+1$. For a tree T, recall the definition of $\mathcal{I}(T)$ in Section 3.3. Given the full binary tree T_t^{pred}, we denote n_0 to be the node that has the largest index in $\mathcal{I}(T_t^{\text{pred}})$ among all $N+1$ internal nodes of T_t^{pred}. It is obvious that both child nodes of n_0 are leaf nodes (if not, say the left child of n_0 is also an internal nodes, then it should have a larger index than n_0, which leads to a contradiction). We use T_0 to denote the tree obtained by removing the two child nodes of n_0 from the tree T_t^{pred} (n_0 is a leaf node of T_0). Then T_0 has exactly N internal nodes. We use $l_1, \ldots, l_{N+2}$ and $l'_1, \ldots, l'_{N+1}$ to denote the lengths of all paths of T_t^{pred} and T_0, respectively. Without loss of generality, we denote the length of the path that ends at n_0 in T_0 as l'_{N+1}, and the length of those two paths that pass through n_0 in T_t^{pred} as l_{N+1} and l_{N+2}, respectively.

Note that $l_i = l'_i, 1 \le i \le N$ and $l_{N+1} = l_{N+2} = l'_{N+1} + 1$. We have

$$\begin{aligned}\sum_{k=1}^{N+2} C_0^{l_k} &= \sum_{k=1}^{N} C_0^{l'_k} + 2C_0^{l'_{N+1}+1} \\ &= \sum_{k=1}^{N+1} C_0^{l'_k} + (2C_0 - 1)C_0^{l'_{N+1}} \\ &\le C_0^N + \sum_{k=1}^{N} C_0^k + (2C_0 - 1)C_0^N \\ &= C_0^{N+1} + \sum_{k=1}^{N+1} C_0^k,\end{aligned}$$

where the inequality follows from the induction equation $\sum_{k=1}^{N+1} C_0^{l'_k} \le C_0^N + \sum_{k=1}^{N} C_0^k$ and the facts that $2C_0 - 1 < 0$ and $l'_{N+1} \le N$. This complete the induction step.

It remains to define the constant ϵ. Since $C_0 < \frac{1}{2}$, we have

$$\begin{aligned}\sum_{k=1}^{N+1} C_0^{l_k} \le C_0^N + \sum_{k=1}^{N} C_0^k &< 2^{-N} + \sum_{k=2}^{N} 2^{-k} + C_0 \\ &= 2^{-N} + \sum_{k=1}^{N} 2^{-k} - \left(\frac{1}{2} - C_0\right) \\ &= 1 - \left(\frac{1}{2} - C_0\right).\end{aligned}$$

Taking $\epsilon = \frac{1}{2} - C_0 > 0$ finishes the proof for Lemma 3. □

By (17) and Lemma 3, for every $t, 1 \le t \le T$ we have

$$\left\| \frac{\partial h_t}{\partial h_{t-1}} \right\| \le \eta \triangleq 1 - \epsilon < 1.$$

Combining with condition (15), we have

$$\left\| \frac{\partial \mathcal{E}_T}{\partial h_T} \left(\prod_{1<t\le T} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial^+ h_1}{\partial \phi} \right\| \le \left\| \frac{\partial \mathcal{E}_T}{\partial h_T} \right\| \prod_{1<t\le T} \left\| \frac{\partial h_t}{\partial h_{t-1}} \right\| \left\| \frac{\partial^+ h_1}{\partial \phi} \right\| \le C_4 \eta^{T-1} \left\| \frac{\partial^+ h_1}{\partial \phi} \right\|.$$

As $\eta < 1$, we have $\left\| \frac{\partial \mathcal{E}_T}{\partial h_T} \left(\prod_{1<t\le T} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial^+ h_1}{\partial \phi} \right\|$ goes to 0 exponentially with $T \to \infty$. □

D.2 Disscussion of Theorem 3

We next discuss the feasibility of conditions (12) – (16). Condition (12) requires all the weight matrices in $\mathcal{L} \cup \mathcal{R}$ to have the spectral norm no larger than C_1. For sigmoid, condition (13) holds for $C_2 = \frac{1}{4}$ while for tanh and ReLU it holds for $C_2 = 1$. Condition (14) bounds the spectral norm of the gradient of each binary function. If the binary operation o is addition, then the Jacobian matrix $\frac{\partial o(L_i v_1, R_i v_2)}{\partial (L_i v_1)}$ is simply the identity matrix and its spectral norm equals to 1. For vector entry-wise multiplication, note that

$$\begin{aligned}\left\| \frac{\partial [(L_i v_1) \odot (R_i v_2)]}{\partial (L_i v_1)} \right\| = \|\mathrm{diag}\{R_i v_2\}\| = \|R_i v_2\|_\infty &\le \|R_i\|_\infty \|v_2\|_\infty \\ &\le \sqrt{p} \|R_i\| \|v_2\|_\infty \le \sqrt{p} C_1 C_5,\end{aligned}$$

where $C_5 \triangleq \max_{v\in\mathcal{V}_t, 1\le t\le T} \|v\|_\infty$ is the upper bound of the infinity norm of all vectors on the predicted trees. Therefore, if $\odot \in \mathcal{O}$, we should have $C_3 \ge \sqrt{p}C_1C_5$. Condition (16) holds when the scale of weight matrices or vectors on nodes of the predicted trees are small. In the experiments we have $C_2 = 1$ and $p = 100$. Note also that $C_5 = 1$ if each v is the outcome of sigmoid or tanh, or entry-wise product of such vectors; in presence of addition this no longer holds however computational experiments have established that $C_5 \le 1$ even if addition is a candidate binary operation. Then condition (16) holds for $C_1 \approx 0.223$ which has been observed in our experiments.

It is worth to mention that condition (15) is mild since we only require the norm to be bounded by a sufficiently large constant. By (6), we have

$$\mathcal{E}_t = \mathbb{E}_{(X,Y)}\big[\lambda_1 l(y_t, q_t) + \lambda_2 \min_{\overline{T}\in \mathrm{Iso}(T_t^{\mathrm{target}})} TD(\overline{T}, T_t^{\mathrm{pred}}) + \lambda_3 \sum_{k=0}^{N-1} m(\mathcal{N}_k^t)\big] + \frac{\lambda_4}{T}\sum_{\phi\in\Phi}\|\phi\|^2 .$$

There are three terms in the expression of $\mathcal{E}_t$ that are related to h_t, namely, the standard loss term, the tree distance term, and the scoring margin term. We bound them one by one. Since the number of samples is finite, we only consider each term for one data point in the following (i.e. we ignore the expectation).

Assume that $\left\|\frac{\partial l(y_t,q)}{\partial q}\right\| \le C_6$, $\left\|\frac{\partial g(x_t,h;\Gamma)}{\partial h}\right\| \le C_7$ hold. Then the loss term is bounded by

$$\left\|\frac{\partial l(y_t,q_t)}{\partial h_t}\right\| = \left\|\frac{\partial l(y_t,q_t)}{\partial q_t}\frac{\partial q_t}{\partial h_t}\right\| \le \left\|\frac{\partial l(y_t,q_t)}{\partial q_t}\right\|\left\|\frac{\partial q_t}{\partial h_t}\right\| = \left\|\frac{\partial l(y_t,q_t)}{\partial q_t}\right\|\left\|\frac{\partial g(x_t,h_t;\Gamma)}{\partial h_t}\right\| \le C_6C_7.$$

If we use h_t^{target} to denote the vector of the root node of the ground truth tree T_t^{target} and assume that $\left\|h_t^{\mathrm{target}}\right\| \le C_8$ for all t. Note that h_t only appears at the root node of the predicted tree T_t^{pred}, and the definition of TD can be regarded as a summation over many norms of vector differences. Then the only term in TD that includes h_t is $\left\|h_t - h_t^{\mathrm{target}}\right\|^2$. Therefore, the tree distance term is bounded by

$$\begin{aligned}\left\|\frac{\partial \mathrm{TD}(T_t^{\mathrm{target}}, T_t^{\mathrm{pred}})}{\partial h_t}\right\| &= \left\|\frac{\partial \left\|h_t - h_t^{\mathrm{target}}\right\|^2}{\partial h_t}\right\| \\ &= 2\left\|h_t - h_t^{\mathrm{target}}\right\| \le 2\left(\|h_t\| + \left\|h_t^{\mathrm{target}}\right\|\right) \le 2(C_5 + C_8).\end{aligned}$$

Again, note that h_t only appears in the term

$$m(\mathcal{N}_{N-1}^t) = \frac{1}{M}\min\Big\{M, \alpha(h_t;\Theta) - \alpha(c_{N-1}^{**};\Theta)\Big\}.$$

If we assume that $\left\|\frac{\partial\alpha(h;\Theta)}{\partial h}\right\| \le C_9$ holds for any vector $h \in \mathbb{R}^p$, then the scoring margin term is bounded by

$$\left\|\frac{\partial \sum_{k=0}^{N-1} m(\mathcal{N}_k^t)}{\partial h_t}\right\| = \left\|\frac{\partial m(\mathcal{N}_{N-1}^t)}{\partial h_t}\right\| \le \left\|\frac{\partial \alpha(h_t;\Theta)}{\partial h_t}\right\| \le C_9.$$

In conclusion, if we assume the existence of constants C_5, C_6, C_7, C_8, and C_9, then the gradient of loss function $\mathcal{E}_t$ with respect to the hidden state h_t is bounded. Note that in practice C_5 and C_8 are about the norm of a finite set of vectors, and C_6, C_7, C_9 bound the norm of some simple functions or networks which in practice are all bounded. Therefore, we can easily argue that these constants do exist and thus the condition (15) is mild.

In summary, gradient vanishing frequently appears in practice.

D.3 Discussion of Theorem 8

In this section, we discuss the conditions appearing in the Theorem 8. As the proof is similar to the proof of Theorem 3 we omit it here.

The conditions listed in Theorem 8 are common in practice since the quantity $(N+1)^{-\frac{1}{3l_{\min}}}$ is smaller than 1. If we have $\tanh \in \mathcal{U}$ or $\mathrm{ReLU} \in \mathcal{U}$, then the first condition above is automatically achieved. Besides, if the addition operation belongs to $\mathcal{B}$, then the third and the fourth conditions are both fulfilled. In summary, gradient exploding is frequent in practice.

E Details of Experimental Study

E.1 Implementation Details

For our implementation of RRNN-GRU, we use PyTorch and train on Nvidia 1080 Ti GPUs or Intel Skylake CPUs. In order for our choices of cell structures to be differentiable with respect to the parameters of the scoring network, we evaluate softmax over the scores of all potential vectors at each node in the cell. Gradient clipping is used for RRNN-GRU, but random hyperparameter search often allows large gradient magnitudes. With the optimal hyperparameters, training of RRNN-GRU takes approximately ten hours for the Wikipedia dataset, one hour for PTB, and eight hours for SST, which is longer than the GRU training time since the RRNN-GRU weights must adapt to multiple placements within the cell structure.

For the RRNN model, we use batch normalization to stabilize training. The RRNN training time on the Wikipedia dataset with 5,000 samples is 40 hours on a CPU of a 12-core server. We find RRNN to be faster on a CPU than GPU due to its structure searching algorithm, but RRNN-GRU's algorithm runs faster on GPUs.

E.2 Hyperparameters

1. GRU on Wiki-5k: batch size of 18, learning rate of 1.71×10^{-3}, and ℓ_2-regularization coefficient of 3.60×10^{-7}.
2. RRNN on Wiki-5k: batch size of 16, learning rate of 10^{-3}, and scoring network hidden size of 256. $\lambda_1 = 1, \lambda_2 = 10^{-3}, \lambda_3 = 10^{-3}, \lambda_4 = 10^{-5}$.
3. GRU on Wiki-10k: batch size of 41, learning rate of 1.4231×10^{-3}, and ℓ_2-regularization coefficient of 1.2124×10^{-11}.
4. RRNN-GRU on Wiki-10k: batch size of 128, learning rate of 10^{-3}, and scoring network hidden size of 64. Training alternates between the L, R, and b weights and the scoring network every five epochs. $\lambda_1 = 1, \lambda_2 = 0.1, \lambda_3 = 10^{-8}, \lambda_4 = 0.003$. Gradients are clipped to the maximum norm of 1.
5. GRU on SST: learning rate of 4.85×10^{-4}, batch size of 3, and ℓ_2 weight decay coefficient of 2.11×10^{-12}.
6. RRNN-GRU on SST: learning rate of 1.06×10^{-5}, $\lambda_1 = 1, \lambda_2 = 1.76 \times 10^{-6}, \lambda_3 = 2.67 \times 10^{-12}, \lambda_4 = 5.47 \times 10^{-5}$, max-margin of 5.47×10^{-5}, scoring network hidden size of 10 nodes, gradients clipped to the norm of 46.3, alternating training every epoch.
7. GRU on PTB: learning rate of 5.29×10^{-4}, batch size of 6, and ℓ_2 weight decay coefficient of 2.71×10^{-15}.
8. RRNN-GRU on PTB: batch size of 116, learning rate of 3.03×10^{-4}, $\lambda_1 = 1, \lambda_2 = 4.16 \times 10^{-3}, \lambda_3 = 1.22 \times 10^{-13}, \lambda_4 = 1.36 \times 10^{-3}$, max scoring margin of 1.74, maximum gradient magnitude of 1.64, scoring hidden size of 137, and alternating training every epoch.

We next list the hyperparamter search ranges for RRNN-GRU in Table 2 and GRU in Table 3.

Table 2: Hyperpameter search range for RRNN-GRU

	Wiki-10k	SST	PTB
Batch size	$[1, 316]$	$[1, 316]$	$[1, 316]$
Learning rate	$[10^{-5}, 10^{-2}]$	$[10^{-5}, 10^{-2}]$	$[10^{-5}, 10^{-2}]$
λ_2	$[10^{-2}, 1]$	$[10^{-5}, 10^{-2}]$	$[3 \times 10^{-2}, 3]$
λ_3	$[10^{-16}, 1]$	$[10^{-5}, 10^{-2}]$	$[3 \times 10^{-16}, 3 \times 10^{-2}]$
λ_4	$[10^{-6}, 10^{-2}]$	$[10^{-8}, 10^{-4}]$	$[3 \times 10^{-6}, 3 \times 10^{-2}]$
Scoring margin M	$[0.1, 10]$	$[0.1, 10]$	$[0.1, 10]$
Gradient clipping threshold	$[0.1, 100]$	$[0.1, 100]$	$[0.1, 100]$
Alternate frequency	$[1, 10]$	$[1, 10]$	$[1, 10]$

Table 3: Hyperpameter search range for GRU

	Wiki-5k/Wiki-10k	SST	PTB
Batch size	$[8, 256]$	$[4, 128]$	$[8, 256]$
Learning rate	$[10^{-5}, 10^{-1}]$	$[10^{-6}, 10^{1}]$	$[10^{-4}, 10^{-1}]$
ℓ_2 weight decay coefficient	$[10^{-16}, 1]$	$[10^{-16}, 10^{-2}]$	$[10^{-16}, 1]$