Temporal Pyramid Recurrent Neural Network

Qianli Ma,[1] Zhenxi Lin,[1] Enhuan Chen,[1] Garrison W. Cottrell[2]
[1]School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
[2]Department of Computer Science and Engineering, University of California, San Diego, CA, USA
qianlima@scut.edu.cn, zhenxi_lin@foxmail.com, ceh930603@gmail.com, gary@ucsd.edu

Abstract

Learning long-term and multi-scale dependencies in sequential data is a challenging task for recurrent neural networks (RNNs). In this paper, a novel RNN structure called temporal pyramid RNN (TP-RNN) is proposed to achieve these two goals. TP-RNN is a pyramid-like structure and generally has multiple layers. In each layer of the network, there are several sub-pyramids connected by a shortcut path to the output, which can efficiently aggregate historical information from hidden states and provide many gradient feedback short-paths. This avoids back-propagating through many hidden states as in usual RNNs. In particular, in the multi-layer structure of TP-RNN, the input sequence of the higher layer is a large-scale aggregated state sequence produced by the sub-pyramids in the previous layer, instead of the usual sequence of hidden states. In this way, TP-RNN can explicitly learn multi-scale dependencies with multi-scale input sequences of different layers, and shorten the input sequence and gradient feedback paths of each layer. This avoids the vanishing gradient problem in deep RNNs and allows the network to efficiently learn long-term dependencies. We evaluate TP-RNN on several sequence modeling tasks, including the masked addition problem, pixel-by-pixel image classification, signal recognition and speaker identification. Experimental results demonstrate that TP-RNN consistently outperforms existing RNNs for learning long-term and multi-scale dependencies in sequential data.

Introduction

Recurrent neural networks (RNNs) are well-known for modeling sequential data by learning temporal dependencies via recurrent connections. They have achieved good performance in various sequence modeling applications, such as pixel-by-pixel image classification (Cooijmans et al. 2017; Trinh et al. 2018; Hu, Qi, and Wang 2019), speech recognition (Tao and Liu 2018), language modeling (Soltani and Jiang 2016; Chung, Ahn, and Bengio 2017) and so on.

RNNs are usually trained by the back-propagation through time (BPTT) method (Werbos 1990). However, it is difficult to train vanilla RNNs to learn long-term dependencies since the gradient tends to either vanish or explode when modeling long sequences. The gradient clipping method (Pascanu, Mikolov, and Bengio 2013) can avoid the exploding gradient problem, while gated RNNs, such as long short-term memory (LSTM) (Hochreiter and Schmidhuber 1997) and gated recurrent units (GRU) (Cho et al. 2014), can alleviate the vanishing gradient problem using linear units with a self-connection weighted by gated units. Besides, additional direct connections from previous time steps to the current time step is another way to improve learning long-term dependencies for RNNs (Soltani and Jiang 2016; Wang 2017; Zilly et al. 2017; Campos et al. 2018). On the other hand, it is a challenge for these single-layer RNNs to learn multi-scale dependencies (Chung, Ahn, and Bengio 2017).

A common strategy to deal with multi-scale sequential data is to design hierarchical information processing systems (Jaeger 2007). Hence, many hierarchical RNNs (Chung et al. 2015; Chang et al. 2017; Chung, Ahn, and Bengio 2017) have been proposed to learn multi-scale dependencies, where various layers in the multi-layer structure specialize on different scales. But multi-layer structures always lead to deep RNNs, which more easily suffer from the gradient vanishing problem and are more difficultly trained to learn long-term dependencies than single-layer RNNs (Li et al. 2018).

In this paper, a novel hierarchical RNN, called temporal pyramid RNN (TP-RNN), is proposed to learn long-term and multi-scale dependencies in sequential data. TP-RNN is a pyramid-like structure and generally has multiple layers. In each layer of the network, there are several sub-pyramids ordered by the time step. The bottom of each sub-pyramid consists of several consecutive hidden states, which are hierarchically aggregated into high-level states that represent larger timescales. Further, the top state nodes of all sub-pyramids in the same layer are iteratively aggregated in time step order through skip connections to form a shortcut path to the output of the layer. The sub-pyramids connected by the shortcut path can efficiently aggregate historical information from hidden states and provide many gradient feedback short-paths.

In particular, in the multi-layer structure of TP-RNN, the input sequence of the higher layer is a large-scale aggregated state sequence produced by the sub-pyramids in the previous layer, but not the usual sequence of hidden states. TP-RNN can therefore explicitly learn multi-scale dependencies with

multi-scale input sequences of different layers, and shorten the input sequence and gradient feedback paths of each layer. This greatly alleviates the vanishing gradient problem in deep RNNs and allows the network to efficiently learn long-term dependencies.

Our main contributions can be summarized as follows:

- We propose a multi-layer RNN structure called TP-RNN consisting of several sub-pyramids connected by a short-cut path to the output in each layer. They can efficiently aggregate historical information from hidden states and provide many gradient feedback short-paths.
- When constructing the multi-layer RNN structure, we use a large-scale aggregated state sequence produced by the sub-pyramids in the previous layer as the input sequence of the higher layer. Hence, TP-RNN can explicitly learn multi- and long-scale dependencies as well as greatly alleviate the vanishing gradient problem.
- We evaluate TP-RNN on several sequence modeling tasks including the masked addition problem, pixel-by-pixel image classification, signal recognition and speaker identification. Our experimental results demonstrate that the TP-RNN consistently performs better than existing RNNs for learning long-term and multi-scale dependencies in sequential data.

Related work

RNNs for learning long-term dependencies In recent years, various methods have been proposed to improve learning long-term dependencies for RNNs, including gating mechanism (Zhou et al. 2016; Bradbury et al. 2017; Jing et al. 2018; Chandar et al. 2019) and special initialization (Le, Jaitly, and Hinton 2015; Arjovsky, Shah, and Bengio 2016; Zhang et al. 2016; Cooijmans et al. 2017; Vorontsov et al. 2017). Addition direct connections from previous time steps to the current time step is also an important method, which allows the gradient to flow back to earlier time steps more efficiently. The first attempt at this is the NARX RNN (Lin et al. 1996), which introduced linear time delayed connections to an RNN structure to make gradient descent learning more effective. Similarly, HORNN (Soltani and Jiang 2016) aggregates multiple preceding hidden states through addition linearly weighted direct connections for better long-term memory. Recently, methods similar to HORNN within the long short-term memory (LSTM) structure have been used for sentiment recognition (Wang 2017; Tao and Liu 2018) and question classification (Xia et al. 2018). Finally, SAB (Ke et al. 2018) adds sparse temporal direct connections to preceding hidden states and aggregates them through an attention mechanism. However, these single-layer RNNs often have difficulties modeling inherent multi-scale structures and learning multi-scale dependencies in sequential data (Chung, Ahn, and Bengio 2017).

Hierarchical multi-scale RNNs In order to learn multi-scale dependencies in sequential data, many hierarchical RNNs have been proposed, where various layers in the multi-layer structure specialize for different scales (Jaeger 2007). The gated-feedback RNN (Chung et al. 2015) adaptively gates the recurrent signals exchanged between layers, which assigns various layers to different scales according to the previous hidden states and current input. Similarly, Kim *et al.* (Kim, Singh, and Lee 2016) proposed a temporal hierarchy based on gated recurrent units that improves learning multi-scale dependencies in large texts. Recently, the Dilated RNN (Chang et al. 2017) employs skip connections through different numbers of time steps in various layers to learn multi-scale dependencies. Chung *et al.* (Chung, Ahn, and Bengio 2017) proposed a hierarchical multi-scale RNN to model latent multi-scale structures in language sequences by a multi-scale updating mechanism. However, these hierarchical RNNs have multiple layers and lead to deep structures, which are subject to vanishing gradients and have more difficulty on learning long-term dependencies compared to single-layer RNNs (Li et al. 2018).

Proposed method

In this section, we propose the temporal pyramid RNN (TP-RNN) for learning long-term and multi-scale dependencies in sequential data. The unfolded computational graph of a TP-RNN is shown in Figure 1. It is a pyramid-like structure and generally has multiple layers. In order to describe the multi-layer structure of a TP-RNN clearly, we will start from the first layer to show the generation process of sub-pyramids in a single layer, and then introduce how to construct a multi-layer structure. Finally, we discuss why this method is effective.

Sub-pyramids

We use $\mathbf{X}$ to denote an input sequence of dimension d_x and length T. As shown in Figure 1, the input sequence is cut into N input subsequences of length $L = T/N$ (T, N and L are 16, 4 and 4 respectively in Figure 1). N is a meta-parameter of our method. The inputs are presented sequentially, driving the generation of a sub-pyramid. Since a sub-pyramid is generated level by level from bottom to top, we define the bottom of a sub-pyramid as Level 0, the next as Level 1, and so on until the top Level J ($J = 2$ in Figure 1).

We now introduce the generation process of a sub-pyramid in detail. We use $\mathbf{H}$ as the initial hidden state sequence of dimension d_h and length T. The bottom of a sub-pyramid is composed of a hidden state subsequence of length L. As indicated by the blue arrows at the bottom of Figure 1 and the red arrows inside a sub-pyramid, the hidden state $\mathbf{h}_l$ at the bottom of a sub-pyramid is driven by the input $\mathbf{x}_l$ of the current time step with the hidden state $\mathbf{h}_{l-1}$ of the previous time step (consistent with conventional RNNs) or a hierarchical aggregated state $\hat{\mathbf{h}}_i^j$ (unlike conventional RNNs). If the hierarchical aggregation operation is performed every g time steps (g is called aggregation granularity and $g = 2$ in Figure 1), the update equation of hidden state $\mathbf{h}_l$ can be expressed as follows:

$$\mathbf{h}_l = \begin{cases} f(\mathbf{U}\mathbf{x}_l + \mathbf{W}\mathbf{h}_{l-1} + \mathbf{b}), & \text{if } (l-1) \not\equiv 0 \pmod{g} \\ f(\mathbf{U}\mathbf{x}_l + \mathbf{W}\hat{\mathbf{h}}_i^j + \mathbf{b}), & \text{if } (l-1) \equiv 0 \pmod{g} \end{cases} \tag{1}$$

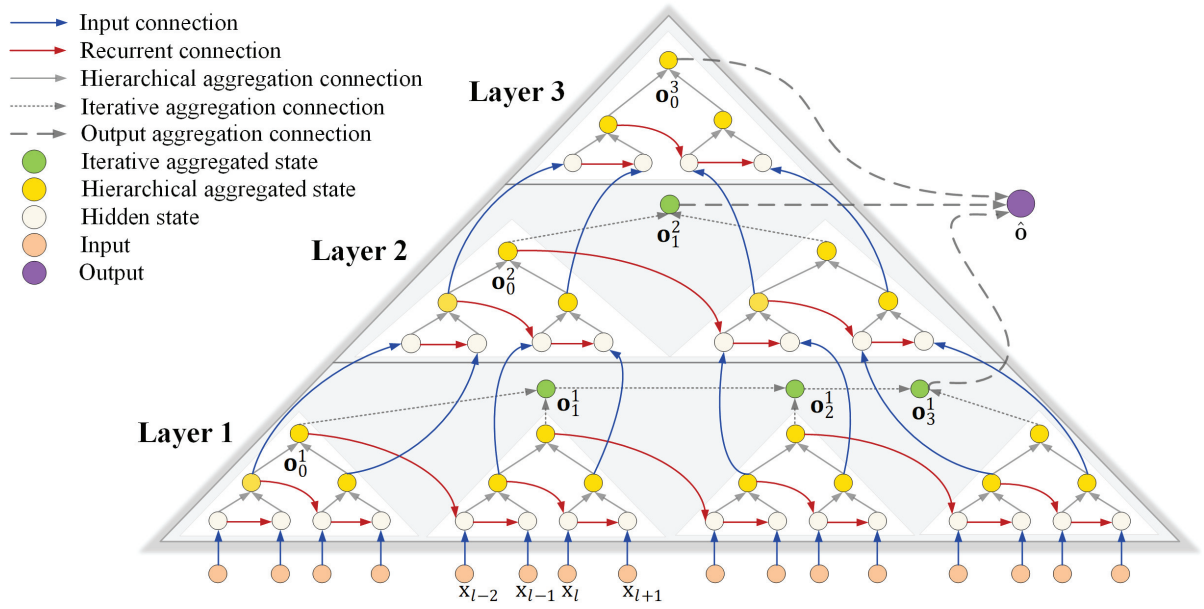

Figure 1: Unfolded computational graph of a TP-RNN. The network is pyramid-like and multi-layer. This example has 3 layers. In each layer, there are several (4, 2 and 1 for Layer 1, 2 and 3 respectively) sub-pyramids ordered by time step. The bottom of each sub-pyramid consists of 4 consecutive hidden states (white nodes), which are hierarchically aggregated into high-level states (yellow nodes) that represent larger timescales. Further, the top (yellow) state nodes of all sub-pyramids in the same layer are iteratively aggregated from left to right through skip connections to form a shortcut path to the output of the layer (the rightmost green node in each layer). In the multi-layer structure, the input sequence of the higher layer is a large-scale aggregated state sequence produced by the sub-pyramids in the previous layer. The final output is obtained by aggregating outputs of all layers. In each layer, all connections with the same color and type use the same weights. All of the aggregation operations use the same function, with different weights depending on the type of aggregation.

where $\mathbf{x}_l$ and $\mathbf{h}_l$ are the l-th input (time step) of an input subsequence and the l-th hidden state of a hidden state subsequence respectively. $f(\cdot)$ denotes the activation function (usually $tanh(\cdot)$), and $\mathbf{U} \in \mathbb{R}^{d_h \times d_x}$, $\mathbf{W} \in \mathbb{R}^{d_h \times d_h}$ and $\mathbf{b} \in \mathbb{R}^{d_h}$ are trainable input weights, recurrent weights and the bias respectively. $\hat{\mathbf{h}}_i^j$ is the i-th hierarchical aggregated state at the j-th level of a sub-pyramid, where j is the number of trailing-zeros of the G-ary number of the value $(l-1)$, and $i = (l-1)/g^j$. Normally, $j \in [1, J]$ where $J = \log_g L$ is the number of levels of a sub-pyramid (not counting Level 0), and $i \in [1, L_j]$ where $L_j = L_{j-1}/g$ ($L_0 = L$) is the number of hierarchical aggregated states at the j-th level of the sub-pyramid. As indicated by the gray arrows inside a sub-pyramid in Figure 1, hierarchical aggregated states are generated level by level through a hierarchical aggregation method, which can be formalized as:

$$\hat{\mathbf{h}}_i^j = \begin{cases} \theta(\mathbf{h}_{(i-1)*g+1}, \mathbf{h}_{(i-1)*g+2}, \cdots, \mathbf{h}_{i*g}), & \text{if } j = 1 \\ \theta(\hat{\mathbf{h}}_{(i-1)*g+1}^{j-1}, \hat{\mathbf{h}}_{(i-1)*g+2}^{j-1}, \cdots, \hat{\mathbf{h}}_{i*g}^{j-1}), & \text{if } 2 \le j \le J \end{cases} \tag{2}$$

where $\theta(\cdot)$ denotes an aggregation function. It aggregates g state nodes (hidden states or hierarchical aggregated states) into a new hierarchical aggregated state using self-attention and sum-reduce operations. The definition of θ will be given later.

Through Equation 1 and 2, all hidden states and hierarchical aggregated states of a sub-pyramid can be successively generated in time step order until the hierarchical aggregated state at the top $\hat{\mathbf{h}}_1^J$ is generated (Level J is the top level of the sub-pyramid). The top state node $\hat{\mathbf{h}}_1^J$ is an information aggregation of an input subsequence. As indicated by the red arrow between two adjacent sub-pyramids in Figure 1, it will drive the generation of the first hidden state at the bottom of next sub-pyramid with the subsequent input. All sub-pyramids in a layer can be generated from bottom to top and left to right in the same way.

After all sub-pyramids in a layer have been generated, all top state nodes are iteratively aggregated in time step order through skip connections (gray dotted arrows above the sub-pyramids in Figure 1) to form a shortcut path (through green nodes in Figure 1) to the output of the layer (the rightmost green node in each layer). This iterative aggregation process can be formalized as:

$$\mathbf{o}_n = \theta(\mathbf{o}_{n-1}, \hat{\mathbf{h}}_{n+1}), n = 1, \cdots, N-1, \tag{3}$$

where $\hat{\mathbf{h}}_{n+1}$ is the top state node $\hat{\mathbf{h}}_1^J$ of the $(n+1)$-th sub-pyramid, and $\mathbf{o}_n$ is the n-th iterative aggregated state of the shortcut path ($\mathbf{o}_0 = \hat{\mathbf{h}}_1$). $\theta(\cdot)$ is the same function as in Equation 2, but uses different weights. $\mathbf{o}_{N-1}$ (N is the number of sub-pyramids of a layer) is the output of the layer, which is an aggregated feature vector of the input sequence $\mathbf{X}$.

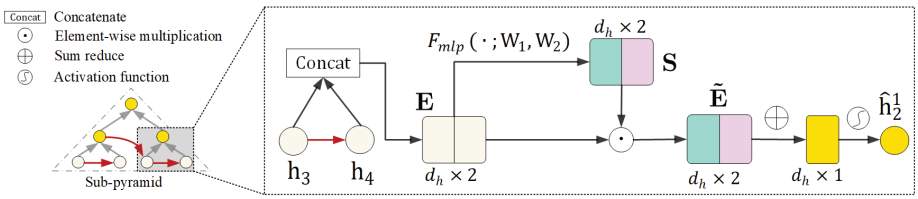

Figure 2: Detailed operations of the aggregation function $\theta(\cdot)$. In this example, it is used to aggregate two hidden states into a hierarchical aggregated state by weighting and summing them.

Multi-layer structure

The previous section has introduced how to construct a single-layer TP-RNN, which corresponds to the bottom trapezoid in Figure 1. Later we will use this to compare performance with a multi-layer TP-RNN. A multi-layer TP-RNN can be constructed by stacking multiple such RNNs to learn multi-scale dependencies in sequential data. As indicated by the blue arrows between two adjacent layers in Figure 1, the stacking method is to use a hierarchical aggregated state sequence produced by the sub-pyramids of the previous layer as the input sequence of the higher layer.

Several hierarchical aggregated state sequences are produced after all sub-pyramids in a layer are generated. Among them, the hierarchical aggregated state sequence at the j-th level of all sub-pyramids can be formalized as:

$$\hat{\mathbf{H}}^j = [\hat{\mathbf{H}}_1^j, \cdots, \hat{\mathbf{H}}_n^j, \cdots, \hat{\mathbf{H}}_N^j], \tag{4}$$

where $\hat{\mathbf{H}}_n^j = [\hat{\mathbf{h}}_1^j, \cdots, \hat{\mathbf{h}}_i^j, \cdots, \hat{\mathbf{h}}_{L_j}^j]$ is a subsequence of hierarchical aggregated states at the j-th level of the n-th sub-pyramid. The hierarchical aggregated state sequence $\hat{\mathbf{H}}^j$ is a high-level and large-scale representation of the input sequence $\mathbf{X}$. Note the difference between using this hierarchical aggregated state sequence $\hat{\mathbf{H}}^j$ (e.g., $j = 1$ in Figure 1) as input to the next layer up and the standard approach of the entire hidden state sequence $\mathbf{H}$ of the previous layer. In general, $\hat{\mathbf{H}}^j$ will be much shorter and more abstract than $\mathbf{H}$.

In each layer, through Equation 1, 2 and 3, we can obtain the outputs of all layers as follows:

$$\mathbf{O} = [\mathbf{o}_{N_1-1}^1, \cdots, \mathbf{o}_{N_k-1}^k, \cdots, \mathbf{o}_{N_K-1}^K], \tag{5}$$

where K and N_k are the number of layers and the number of sub-pyramids in the k-th layer respectively, and $\mathbf{o}_{N_k-1}^k$ is the output of the k-th layer. The output set $\mathbf{O}$ contains features on K scales of the input sequence $\mathbf{X}$. Finally, the output of a TP-RNN is obtained by the aggregation function $\theta(\cdot)$ based on the output set $\mathbf{O}$ as follows:

$$\hat{\mathbf{o}} = \theta(\mathbf{O}) = \theta(\mathbf{o}_{N_1-1}^1, \cdots, \mathbf{o}_{N_k-1}^k, \cdots, \mathbf{o}_{N_K-1}^K). \tag{6}$$

The output $\hat{\mathbf{o}}$ is a multi-scale fusion feature of the input sequence $\mathbf{X}$. After, we can simply use a SoftMax classifier to do classification tasks and train the network using BPTT.

Aggregation function

In this section, we formally define the aggregation function $\theta(\cdot)$ applied in Equation 2, 3 and 6. It is used to aggregate multiple state nodes (hidden states, hierarchical or iterative aggregated states) into a new state node by weighting and summing them. Its detailed operations are shown in Figure 2.

Given M d_h-dimensional state vectors (e.g., $\mathbf{h}_l$, $\hat{\mathbf{h}}_i^j$ or $\mathbf{o}_{N_k-1}^k$) to be aggregated, we can concatenate them into a state matrix by column as follows:

$$\mathbf{E} = [\mathbf{e}_1, \cdots, \mathbf{e}_m, \cdots, \mathbf{e}_M], \tag{7}$$

where $\mathbf{e}_m \in \mathbb{R}^{d_h}$ is the m-th state vector and $\mathbf{E} \in \mathbb{R}^{d_h \times M}$ is the state matrix. Based on $\mathbf{E}$, we use a one-hidden layer perceptron $F_{mlp}(\cdot)$ to obtain attention weights as follows (similar to the self-attention mechanism of SENet (Hu, Shen, and Sun 2018) but with *fine-grained* effects, we will explain later):

$$\mathbf{S} = F_{mlp}(\mathbf{E}; \mathbf{W}_1, \mathbf{W}_2) = f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{E}^{\mathrm{T}}))^{\mathrm{T}}, \tag{8}$$

where $f_1(\cdot)$ and $f_2(\cdot)$ are nonlinear activation functions ($ReLU(\cdot)$ and $Sigmoid(\cdot)$ respectively), and $\mathbf{W}_1 \in \mathbb{R}^{D \times M}$ and $\mathbf{W}_2 \in \mathbb{R}^{M \times D}$ are trainable weights, and $\mathbf{S} \in \mathbb{R}^{d_h \times M}$ is the self-attention weight matrix. Note that, $\mathbf{W}_1$ and $\mathbf{W}_2$ are shared among time steps just like other parameters of a RNN, but differ among the three aggregation functions in Equation 2, 3 and 6. We then obtain the weighted state matrix:

$$\tilde{\mathbf{E}} = \mathbf{S} \odot \mathbf{E}, \tag{9}$$

where $\odot$ denotes the element-wise multiplication and $\tilde{\mathbf{E}} \in \mathbb{R}^{d_h \times M}$ is the weighted state matrix. Finally, for each dimension of $\tilde{\mathbf{E}}$, we sum up all the M elements row-wise and apply an activation function $f(\cdot)$ (usually $tanh(\cdot)$) to the sum to obtain a new state vector as follows:

$$\hat{\mathbf{e}} = \theta(\mathbf{E}) = (\hat{e}^1, \cdots, \hat{e}^d, \cdots, \hat{e}^{d_h})^{\mathrm{T}}, \text{where } \hat{e}^d = f(\sum_{m=1}^{M} \tilde{e}_m^d), \tag{10}$$

where $\hat{\mathbf{e}} \in \mathbb{R}^{d_h}$ is the aggregated state vector and $\hat{e}^d$ is its scalar element of the d-th dimension, and $\tilde{e}_m^d$ is the scalar element at the d-th row and the m-th column of $\tilde{\mathbf{E}}$.

The main differences between our aggregation method and the one used by SENet are Equations 8-9. In SENet, global average pooling is used to gather global information for each state vector, and the resultant single global feature is distributed to all the elements (locations) of the state vector (*coarse-grained*). Since each element of the state vector is an individual feature, they should have different attention weights. Therefore, we used the state matrix $\mathbf{E}$ instead of the single global feature vector to obtain a weight matrix $\mathbf{S}$, which is then distributed to each element of the state matrix (*fine-grained*). The number of parameters used in this way is consistent with the number in SENet, while our results are better (see Model Analysis).

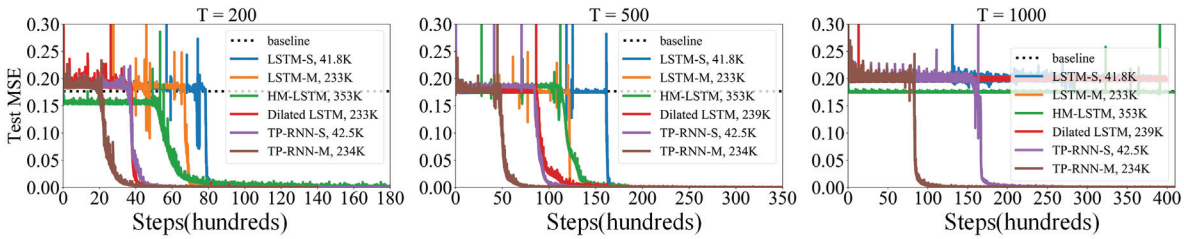

Figure 3: The test MSE curves on the masked addition problem for three sequence length, T = 200 (left), 500 (center) and 1000 (right). #K represents the approximate number of parameters.

Effectiveness discussion

First, compared with conventional RNNs, the sub-pyramids with the shortcut path in each layer of a TP-RNN provide many gradient feedback short-paths, and the length of the longest gradient feedback path from inputs to the output is shortened to the number of levels of the sub-pyramids ($J = \log_g L$) plus the length of the shortcut path ($N = T/L$), which is always much shorter than the input sequence length as in conventional RNNs (i.e., $J + N << T$).

Second, when constructing the multi-layer structure of a TP-RNN, we don't use the usual sequence of hidden states as the input sequence of the higher layer. Instead, we use the hierarchical aggregated state sequence of the previous layer, since it is a high-level and large-scale representation of the input sequence and its length (i.e., $L_j * N = T/g^j$) is always much shorter than that of the hidden state sequence (i.e., $T/g^j << T$). In this way, in the multi-layer structure of a TP-RNN, the higher the layer, the shorter the input sequence length and the longer the input sequence scale.

As a result, TP-RNN can learn multi-scale dependencies with multi-scale input sequences of different layers, as well as shorten the input sequence and gradient feedback paths of each layer, which greatly alleviates the vanishing gradient problem. We also discuss the advantages of TP-RNN over existing hierarchical RNNs (Chung et al. 2015; Kim, Singh, and Lee 2016; Chung, Ahn, and Bengio 2017; Chang et al. 2017) in the Supplementary material.

Meanwhile, TP-RNN is still a light model compared with conventional RNNs, since the only additional parameters are $\mathbf{W}_1$ and $\mathbf{W}_2$ in the aggregation function $\theta(\cdot)$ and they are also shared among time steps just like other parameters of a RNN. Hence, the parameter size of a TP-RNN still mostly depends on the hidden size and the number of layers.

Experiments

We evaluate TP-RNN on several sequence modeling tasks including the masked addition problem, pixel-by-pixel image classification, signal recognition and speaker identification. Experimental results demonstrate that TP-RNN consistently outperforms existing RNNs for learning long-term and multi-scale dependencies in sequential data.

In our experiments, LSTM is used as the basic RNN unit for TP-RNN. We mainly compare TP-RNN with three strong baselines, LSTM, HM-LSTM (Chung, Ahn, and Bengio 2017) and Dilated LSTM (Chang et al. 2017). We name single-layer and multi-layer networks with -S and -M tags, respectively. Dilated LSTMs usually need to stack many layers to achieve good results. The number of layers for LSTM-M, HM-LSTM and TP-RNN-M are both set to 3, and the hidden sizes for LSTMs, HM-LSTM and TP-RNNs are both set to 100. For Dilated LSTMs, the number of layers is set to 9 as in (Chang et al. 2017), while the hidden size is set to 59 for a comparable number of parameters with LSTM-M, HM-LSTM and TP-RNN-M (this setting yields better results than those in (Chang et al. 2017)). All models are trained with the Adam optimizer and the learning rate and decay rate are set to 1e-3 and 0.9, respectively. Notably, we did not use any regularization techniques such as dropout or layer normalization. More detailed discussion of hyper-parameters for all tasks can be found in the Supplementary material[1].

Masked addition problem

The masked addition problem (Hochreiter and Schmidhuber 1997) is a synthetic task designed to evaluate the performance of RNNs for learning long-term dependencies. The input is two sequences of length T. The first sequence is randomly sampled from a uniform distribution in $[0, 1]$, while the second is all 0's except for two 1's in random locations. The target is the sum of the two elements in the first sequence indicated by the two 1's in the second. A baseline is to always predict the target to be a value of 1 regardless of the samples, which gives an MSE around 0.1767. The goal is to train a model achieving MSE well below 0.1767. The masked addition problem becomes harder as the length T increases. In our experiments, three different sequence lengths, T = 200, 500 and 1000, are used to verify the models.

The results are shown in Figure 3. When the sequence length is relatively short (i.e., T = 200 and 500), all of the models converge to a very low MSE. Among them, TP-RNN-M converges first. The performance of TP-RNN-S is faster than HH-LSTM and very close to Dilated LSTM, but TP-RNN-S has only about one-eighth and one-fifth of the number of parameters. TP-RNNs converge much faster than LSTMs with similar numbers of parameters. When the sequence length increases to 1000, neither LSTMs nor HM-LSTM and Dilated LSTM converge to a MSE lower than the baseline, while TP-RNNs still quickly converge to a very low MSE. TP-RNNs clearly overcome the vanishing gradient

[1]The Supplementary material is publicly available at https://github.com/qianlima-lab/TPRNN

problem to learn long-term dependencies.

Pixel-by-pixel image classification

We evaluate TP-RNN on two pixel-by-pixel image classification data sets, MNIST and pMNIST (Le, Jaitly, and Hinton 2015). The pixels of each image are fed into RNNs sequentially for classification. pMNIST is a harder version of MNIST (Arjovsky, Shah, and Bengio 2016), since its pixel sequences are perturbed by a fixed random permutation, which breaks local structure and creates more complex dependencies on various time scales (Trinh et al. 2018). They have become the most popular benchmarks for evaluating long-term dependency learning.

TP-RNN-M exceeds the performance of several previous state-of-the-art RNNs (Table 1). TP-RNN-S achieves comparable performance using many fewer parameters than Dilated LSTM and HM-LSTM. Its superior performance on pMNIST over LSTMs is significant, since pMNIST is more difficult. Moreover, multi-layer LSTM and TP-RNN performing better than single-layer nets prove the benefit of learning multi-scale dependencies in sequential data with multiple layers.

TP-RNNs again converge much faster than LSTMs and Dilated LSTMs on these tasks. In addition, we also evaluate TP-RNN on a more challenging task called noisy MNIST (Chang et al. 2017), which pads the pixel sequences with random noise sampled from a uniform distribution in $[0, 1]$ to the length of T. We use three setups, T = 1000, 2000 and 3000, and get a similar conclusion to the masked addition problem. Details are in the Supplementary material.

Table 1: Classification accuracies (%) on MNIST and pMNIST data sets.

Model	#Params	MNIST	pMNIST
iRNN (Le, Jaitly, and Hinton 2015)	–	97.0	82.0
uRNN (Arjovsky, Shah, and Bengio 2016)	–	95.1	94.0
Stanh-RNN (Zhang et al. 2016)	–	98.1	94.0
BN-LSTM (Cooijmans et al. 2017)	–	98.1	94.0
SAB (Ke et al. 2018)	–	–	94.2
ASLSTM (Hu, Qi, and Wang 2019)	–	98.3	90.8
r-LSTM (Trinh et al. 2018)	–	98.4	95.2
IndRNN (Li et al. 2018)	–	99.0	96.0
LSTM-S	41.8K	98.0	91.3
LSTM-M	233K	98.6	92.3
HM-LSTM (Chung, Ahn, and Bengio 2017)	353K	98.6	94.6
Dilated LSTM (Chang et al. 2017)	239K	99.0	95.9
TP-RNN-S	42.4K	98.7	96.0
TP-RNN-M	234K	**99.2**	**96.7**

Signal recognition

To further evaluate TP-RNN for learning long-term and multi-scale dependencies, we generate a synthetic signal data set inspired by (Hu, Qi, and Wang 2019). It contains 9000 sequences of length 1000, which are divided into a training set, a validation set and a test set according to the ratio of 7:1:2. Each sequence contains n signals of the same type and these signals may have different frequencies (i.e., timescales) f. The number, frequencies and positions of signals are random with $n \in \{3, 4, 5\}$ and $f \in \{20, 40, 60, 80\}$, and the signals do not overlap. The rest of the sequences are filled with noise randomly sampled from a uniform distribution in $[-1, 1]$. A example is shown in Figure 4.

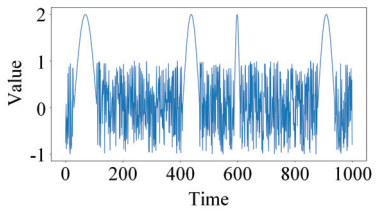

Figure 4: An example of synthetic signal sequence of length 1000. It contains 4 sine signals whose frequencies are 80, 60, 20 and 60 from left to right respectively.

First, we do the task called Signal Type Identification (STI) proposed by (Hu, Qi, and Wang 2019). It requires distinguishing the type of signals of a sequence containing limited useful information. There are three kinds of signal types including sine wave, square wave and saw-tooth wave. Table 2 shows the identification accuracies on STI. This task is relatively easy and most of the models can achieve good performance except LSTM-M, since it is hard to train the deep LSTM structure for learning long-term dependencies.

In fact, the first task does not significantly require learning multi-scale dependencies since it only concerns about the type of signals while ignoring the frequencies. So we design another task called Signal Frequency Counting (SFC) based on the same data set. It requires counting the number of kinds of signal frequencies of a sequence. For example, the signals of the sequence in Figure 4 have three kinds of frequencies, so the label is 3.

The counting accuracies of SFC are also shown in table 2. On this harder task, LSTMs perform very badly while TP-RNNs, especially TP-RNN-M, still perform quite well. TP-RNN-S again achieves better performance than Dilated LSTM and HM-LSTM with many fewer parameters. The sequences are long (length 1000) with a small amount of signal and a lot of noise, and the SFC task requires to deal with different frequencies (i.e., timescales). Hence, this further indicates the effectiveness of TP-RNN for learning long-term and multi-scale dependencies in sequential data.

Table 2: Accuracies (%) of STI and SFC.

Model	#Params	STI	SFC
LSTM-S	41.8K	92.8	50.2
LSTM-M	233K	43.7	55.1
HM-LSTM (Chung, Ahn, and Bengio 2017)	353K	99.6	86.1
Dilated LSTM (Chang et al. 2017)	239K	**100**	76.7
TP-RNN-S	42.4K	99.6	86.5
TP-RNN-M	234K	**100**	**99.8**

Speaker identification

We also evaluate TP-RNN on a speaker identification task using the English multi-speaker corpus from CSTR voice cloning toolkit (VCTK) (Yamagishi 2012). This data set consists of 44 hours of audio from 109 different speakers. Each

Table 3: The identification accuracies (%) on the VCTK data set. "/" represents that the model failed to converge. Acc (#Hz) represents the accuracy of raw audio waves with #Hz sampling rate.

Model	MFCC		Raw		
	#Params	Acc	#Params	Acc (1000Hz)	Acc (1500Hz)
LSTM-S	56.6K	92.3	51.8K	/	/
LSTM-M	247K	95.8	243K	/	/
HM-LSTM (Chung, Ahn, and Bengio 2017)	368K	93.8	364K	/	/
Dilated LSTM (Chang et al. 2017)	248K	96.0	246K	90.7	89.8
TP-RNN-S	56.9K	96.5	53K	88.9	91.0
TP-RNN-M	248K	**97.9**	245K	**93.9**	**95.5**

speaker reads out about 400 sentences. We follow the data splitting protocol, 7:1:2, for each speaker to get the training, validation and test sets. First, we compute 13-dimensional log-mel frequency features (MFCC) with 25ms window and 10ms shift, resulting in sequences of length about 300. Besides, we also directly employ the raw audio waves with 1000Hz sampling rate as input, resulting in very long sequences of length about 3600. Such a sequence length is very challenging for training RNNs.

The results are shown in Table 3. TP-RNNs outperform LSTMs with an accuracy gain over 4% when using MFCC features. And TP-RNN-S still achieves a better result with many fewer parameters than Dilated LSTM and HM-LSTM. Furthermore, when using raw audio waves, LSTMs and HM-LSTM fail to converge since the sequences are too long and they suffer from the vanishing gradient problem during training. TP-RNNs still converge to high accuracy and TP-RNN-M is consistently better than Dilated LSTM. Finally, we increased the sampling rate to 1500Hz to result in sequences of length about 5400. This shows that TP-RNNs can benefit from longer sequences to reach a higher accuracy close to that of MFCC features, while the performance of Dilated LSTM drops significantly. These results demonstrate the ability of the TP-RNN for learning long-term dependencies.

Model Analysis

Impact of aggregation methods We compare three aggregation methods (mean pooling, max pooling and SENet) and test on MNIST and PMNIST. Here, "SENet" doesn't mean the SENet model, it means we use the same aggregation method as SENet did. The results in table 4 show that our method is consistently better than SENet because of fine-grained effects and mean pooling is the worst, while the performance of max pooling is close to SENet.

Table 4: The result of different aggregation methods on MNIST and PMNIST

Task	Model	Mean Pool	Max Pool	SENet	Ours
MNIST	TP-RNN-S	97.8	98.1	98.3	**98.7**
	TP-RNN-M	98.8	99.0	99.1	**99.2**
PMNIST	TP-RNN-S	94.4	95.2	95.6	**96.0**
	TP-RNN-M	95.8	96.2	96.0	**96.7**

Impact of model structures To investigate the effect of each structure in our model, we conduct a set of ablation experiments on MNIST and PMNIST. w/o IAC means we remove the iterative aggregation connections. w/o TDC means we remove the top-down connections between sub-pyramids. w/o OAC means we remove the output aggregation connections. For w/o OAC, we use the output of the last layer to classify. Table 5 shows that when the IAC is removed, the performance is significantly reduced because of the longer feedback path, removing the TDC will make the information flow discontinuous and also reduce the performance, and removing the OAC also affect performance. This proves that all structures are helpful to our model.

Table 5: The ablation test for model structures on MNIST and PMNIST

Task	Model	w/o IAC	w/o TDC	w/o OAC	Ours
MNIST	TP-RNN-S / -M	97.8 / 98.6	98.4 / 99.0	– / 99.0	**98.7 / 99.2**
PMNIST	TP-RNN-S / -M	95.1 / 96.0	95.5 / 96.1	– / 96.2	**96.0 / 96.7**

Conclusion

In this paper, we propose a novel hierarchical RNN named temporal pyramid RNN (TP-RNN). Its structure allows it to learn long-term and multi-scale dependencies in sequential data. The sub-pyramids connected by a shortcut path to the output in each layer of a TP-RNN can efficiently aggregate historical information from hidden states and provide many gradient feedback short-paths. In the multi-layer version of TP-RNN, the input sequence of the higher layer is a large-scale aggregated state sequence produced by the sub-pyramids in the previous layer. In this way, TP-RNN can learn multi-scale dependencies with multi-scale input sequences of different layers, as well as effectively learn long-term dependencies by shortening the input sequence and gradient feedback paths of each layer. TP-RNN achieves better performance than existing RNNs for learning long-term and multi-scale dependencies on several sequence modeling tasks. Our future work will aim to adaptively construct the sub-pyramids according to latent sequence properties.

Acknowledgments

We thank the anonymous reviewers for their helpful feedbacks. The work described in this paper was partially funded by the National Natural Science Foundation of China (Grant Nos. 61502174, 61872148), the Natural Science Foundation of Guangdong Province (Grant Nos.

2017A030313355, 2017A030313358, 2019A1515010768), the Guangzhou Science and Technology Planning Project (Grant Nos. 201704030051, 201902010020).

References

Arjovsky, M.; Shah, A.; and Bengio, Y. 2016. Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, 1120–1128.

Bradbury, J.; Merity, S.; Xiong, C.; and Socher, R. 2017. Quasi-recurrent neural networks. In *International Conference on Learning Representations*.

Campos, V.; Jou, B.; Giró i Nieto, X.; Torres, J.; and Chang, S. 2018. Skip rnn: learning to skip state updates in recurrent neural networks. In *International Conference on Learning Representations*.

Chandar, S.; Sankar, C.; Vorontsov, E.; Kahou, S. E.; and Bengio, Y. 2019. Towards non-saturating recurrent units for modelling long-term dependencies. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, 3280–3287.

Chang, S.; Zhang, Y.; Han, W.; Yu, M.; Guo, X.; Tan, W.; Cui, X.; Witbrock, M.; Hasegawa-Johnson, M.; and Huang, T. S. 2017. Dilated recurrent neural networks. In *Advances in Neural Information Processing Systems*, 77–87.

Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of EMNLP*.

Chung, J.; Ahn, S.; and Bengio, Y. 2017. Hierarchical multiscale recurrent neural networks. In *International Conference on Learning Representations*.

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2015. Gated feedback recurrent neural networks. In *International Conference on Machine Learning*, 2067–2075.

Cooijmans, T.; Ballas, N.; Laurent, C.; Gülçehre, Ç.; and Courville, A. 2017. Recurrent batch normalization. *International Conference on Learning Representations*.

Hochreiter, H., and Schmidhuber, J. 1997. Long short-term memory. *Neural Computation* 9(8):1735–1780.

Hu, H.; Qi, G.; and Wang, L. 2019. Learning to adaptively scale recurrent neural networks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, 3822–3829.

Hu, J.; Shen, L.; and Sun, G. 2018. Squeeze-and-excitation networks. In *Computer Vision and Pattern Recognition*, 7132–7141.

Jaeger, H. 2007. Discovering multiscale dynamical features with hierarchical echo state networks. *Vtls Inc* 35(2):277–284.

Jing, L.; Gulcehre, C.; Peurifoy, J.; Shen, Y.; Tegmark, M.; Soljacic, M.; and Bengio, Y. 2018. Gated orthogonal recurrent units: On learning to forget. In *AAAI Workshops*.

Ke, N. R.; Goyal, A.; Bilaniuk, O.; Binas, J.; Mozer, M. C.; Pal, C.; and Bengio, Y. 2018. Sparse attentive backtracking: Temporal credit assignment through reminding. In *Advances in Neural Information Processing Systems*, 7651–7662.

Kim, M.; Singh, D. M.; and Lee, M. 2016. Towards abstraction from extraction: Multiple timescale gated recurrent unit for summarization. In *Proceedings of the 1st Workshop on Representation Learning for NLP*, 70–77.

Le, Q. V.; Jaitly, N.; and Hinton, G. E. 2015. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*.

Li, S.; Li, W.; Cook, C.; Zhu, C.; and Gao, Y. 2018. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 5457–5466.

Lin, T.; Horne, B. G.; Tino, P.; and Giles, C. L. 1996. Learning long-term dependencies in narx recurrent neural networks. *IEEE Transactions on Neural Networks* 7(6):1329–1338.

Pascanu, R.; Mikolov, T.; and Bengio, Y. 2013. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 1310–1318.

Soltani, R., and Jiang, H. 2016. Higher order recurrent neural networks. *arXiv preprint arXiv:1605.00064* 1.

Tao, F., and Liu, G. 2018. Advanced lstm: A study about better time dependency modeling in emotion recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2906–2910.

Trinh, T. H.; Dai, A. M.; Luong, M.-T.; and Le, Q. V. 2018. Learning longer-term dependencies in rnns with auxiliary losses. In *International Conference on Machine Learning*, 4965–4974.

Vorontsov, E.; Trabelsi, C.; Kadoury, S.; and Pal, C. 2017. On orthogonality and learning recurrent networks with long term dependencies. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, 3570–3578.

Wang, C. 2017. Rra: recurrent residual attention for sequence learning. *arXiv preprint arXiv:1709.03714*.

Werbos, P. J. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* 78(10):1550–1560.

Xia, W.; Zhu, W.; Liao, B.; Chen, M.; Cai, L.; and Huang, L. 2018. Novel architecture for long short-term memory used in question classification. *Neurocomputing* 299:20–31.

Yamagishi, J. 2012. CSTR VCTK corpus: English multi-speaker corpus for CSTR voice cloning toolkit. http://homepages.inf.ed.ac.uk/jyamagis/page3/page58/page58.html.

Zhang, S.; Wu, Y.; Che, T.; Lin, Z.; Memisevic, R.; Salakhutdinov, R. R.; and Bengio, Y. 2016. Architectural complexity measures of recurrent neural networks. In *Advances in Neural Information Processing Systems*, 1822–1830.

Zhou, G.-B.; Wu, J.; Zhang, C.-L.; and Zhou, Z.-H. 2016. Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing* 13(3):226–234.

Zilly, J. G.; Srivastava, R. K.; Koutník, J.; and Schmidhuber, J. 2017. Recurrent highway networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, 4189–4198.