Progressively Stacking Differentiable Architecture Search (PS-DARTs) for Recurrent Neural Networks (RNNs)

By

Yubo Du

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

December 18, 2021

Nashville, Tennessee

Approved:

Gautam Biswas, Ph.D

Marcos, Quinones Grueiro, Ph.D

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

Introduction

Multivariate time series data is important in our daily life ranging from stock prices, traffic occupancy of highways and electricity consumption of buildings, etc. Predicting future information based on historical observations helps people allocate resources properly. For example, based on the prediction of hourly traffic occupancy for different regions, the traffic control center can divert the traffic flow of the popular highways and distribute the police for different districts before the peak hours. With an accurate prediction, congestion can be alleviated and potential accidents can be reduced.

To solve the multivariate time series prediction (MTSP) problems, quite a lot of ideas have been proposed, which can be divided into traditional methods [1, 2, 3, 4, 5] and deep learning approaches [6, 7, 8, 9, 10, 11]. In the traditional methods, a large body of work is based on the idea of auto-regression models which fit previous observations and future information into regression equations. Although these methods are simple yet efficient, they can not always capture short/long-term repetitive patterns in time series. Application data in the real-world usually shows cycles related to the hour of the day, the day of a week, the month in the year, etc. To improve this ability, deep learning approaches such as recurrent neural network (RNN) based long short-term memory model (RNN-LSTM) [12, 13, 14] and Gated Recurrent Unit (RNN-GRU) [7], convolution neural networks (CNN) [9] are proposed. Recently deep learning models mentioned above were combined with attention models to further improve their abilities on capturing complicated repetitive patterns [11, 10]. However, these manually designed models required much human effort and time on adjusting the architectures and performed worse on data without repetitive patterns than traditional approaches.

Neural Architecture Search (NAS) was first introduced to solve the time and human ef-

forts in manually designing deep learning models. In the early work [15], the reinforcement learning (RL) or evolutionary agents [16] was used to evaluate and predict the performance of architectures by taking validation loss as the reward after each training cycle. However, these approaches demand enormous computing resources due to the numerous training and validation cycles required to learn. To improve the efficiency of NAS, Differentiable Architecture Search (DARTs) [17] was proposed. In this work, the discrete architecture of RNN such as the connections of layers and choice of activation functions were relaxed to a continuous space so that they could be updated through gradient descent. With this framework, the controller was not needed anymore and the search can be done in one day with only one GPU.

When applying DARTs to design RNNs for the MTSP tasks, we found that their search space was constrained to the recurrent highway network (RHN) [18] architecture, which limited the ability to capture dynamics in data. As shown in Fig. 3.1(a), the search space of a single RNN layer in DARTs is the same as RHN except for one activation labeled as f was decided by the search.

In our work, we extend the search space of RNNs more than millions larger than DARTs and design a progressive stacking approach that enables the search to finish in a competitive time. As shown in Fig. 3.1(b), we consider every activation function, the input components for each activation function, and the decision about whether to tie the input between two gates to be searchable. With this design, the search space is increased by $2e85$ which is proved in section 3.1.1.4 and the architecture is designed properly for the data either with or without repetitive patterns. To avoid memory and searching time explosion brought by the large search space, we further design a progressively stacking framework that conducts the search in a one-by-one manner for each node instead of searching all of them at once.

The contributions of our work can be summarized as followings:

- **Larger Search Space**: We relax the constraints on the search space by adding all activation functions, linear layers and decision on whether to tie the input for gates

into the search. Thus, we are able to explore architectures that fit better for each specific task.

- **Less Hyperparameter Tuning**: The architectures with limited hyperparameter search achieve better or competitive results compared with the manually designed deep learning models with exhaustive hyperparameter search.

The remainder of this paper is organized as follows. In chapter 2 we review related work and background knowledge. Then, in chapter 3 we describe the proposed method. Next, we present and analyze our approach on three real world time series datasets in chapter 4. Finally, we conclude in chapter 5.

Chapter 2

Background and Related Work

In this part, we first presents the definition of MTSP. Then, we introduces the traditional approaches and deep learning models that have ever been applied on this problem. After that, we explains the background knowledge about DARTs where PS-DARTs derives from.

2.1 Problem Formulation

In MTSP task, given a time series data $X = x_1, x_2, ..., x_{t-1}$, where $x_i \in R^n$, n is the number of features, we want to predict the information at x_{t-1+h}, where h is a fixed horizon. We denote this prediction as y_{t-1+h}, and the ground-truth value as $\hat{y}_{t-1+h} = x_{t-1+h}$. Moreover, for every task, we make an assumption that there is no useful information before the window thus we can use only $x_{t-w}, x_{t-w+1}, ..., x_{t-1}$ to predict x_{t-1+h}, where w is the window size.

2.2 Traditional Approaches

Traditional approaches for time series prediction tasks can be further divided into linear and non-linear models. The most popular linear model is the autoregressive integrated moving average (ARIMA) [1] which incorporated other autoregressive time series models, such as AR, moving average (MA), and autoregressive moving average (ARMA). Linear support vector regression (SVR) [19] model was also designed to solve time series forecasting problems by treating them as typical regression problems with time-varying parameters. However, these models are limited to only linear univariate time series. For solving MTSP, a generalization of AR-based model called vector autoregression (VAR) [20] was proposed. To promote the abilities to capture the nonlinear properties for MTSP tasks, one of the most popular approaches based on kernel methods called gaussian process (GP) [21] was come

up with. However, all the approaches mentioned above failed to capture the complicated features such as repetitive patterns.

2.3 Deep Learning Models

RNNs have been used widely for time series forecasting because of their abilities on exhibiting temporal dynamic behaviors. Among them, the most popular variants are long/short-term memory models (LSTMs) [12, 13, 14] and gated recurrent unit (GRU) [7] whose success was credited to the tricky designing of the gates. For example, to capture the information in a long series of data efficiently, GRU implemented a reset gate to deal with which short-term memory should be kept and an update gate for deciding which long-term memory should be deleted. When applying them to real-world data with different characters, we need further adjust the activation functions of each gate or connections among gates to better express different properties. But manually adjusting these structures took a lot of time due to the larger numbers of the possible combinations. In our work, we overcame this problem by using NAS to automatically design the appropriate RNN representations in several GPU hours.

Long-and short-term temporal pattern network (LSTNet) [10] was a recent GRU-based approach. It added a CNN layer which lowered down the dimension of features followed by the GRU layer with skip connections. A temporal attention module between distant cells along with the local connections between adjacent cells was also added to improve the models' ability to capture more important features.

Another representation work was temporal pattern attention-based LSTM (TPA-LSTM) [11]. It proposed a set of filters to extract time-invariant temporal patterns. Then it combined an attention mechanism that selected the relevant variables as opposed to the relevant time steps with the LSTM model.

Although these two models achieved state-of-the-art accuracy in real-world datasets, they showed less power for dealing with the data without repetitive patterns and need man-

ual efforts in designing the architecture.

2.4 NAS

Different from previous models, NAS aims to design the deep learning model automatically. Thus, the definition of the search space where the search is conducted is important.

In NAS, the RNN cell at time step t is defined as a directed acyclic graph (DAG) consists of N ordered nodes as shown in Fig. 2.1. To avoid the cycles in the graph, a node should only accept the output from the nodes with lower indexes as it input. For the definition of the architecture inside each node, recent works [22, 17, 23] take each node as one layer of recurrent highway network (RHN) [18] as Eq. 2.1 - 2.2 for the first node in the cell and Eq. 2.3 - 2.4 for the other nodes, where $x^{(t)}$ is the RNN signal x at its recurrent time step t; $h_l^{(t)}$ is the output of node l at time t; for $l = 2,3,...,N$, node h_ℓ receives its input from a layer $j_\ell \in h_1,...,h_{\ell-1}$; $W^{(x,c)}$, $W_{\ell,j_\ell}^{(c)}$, $W_0^{(c)}$,$W^{(x,h)}$, $W_{\ell,j_\ell}^{(h)}$, $W_1^{(h)}$ are parameters in linear layers. Only the activation function f_ℓ as well as the connections among nodes j_ℓ in the Eq. 2.4 attends into the search, all the other parts like $c_1^{(t)}$, $h_1^{(t)}$ and $c_\ell^{(t)}$ keeps unchanged.

$$\boldsymbol{c}_1^{(t)} = sigmoid(\boldsymbol{x}^{(t)}\boldsymbol{W}^{(\boldsymbol{x},\boldsymbol{c})} + \boldsymbol{h}_N^{(t-1)}\boldsymbol{W}_0^{(\boldsymbol{c})}) \tag{2.1}$$

$$\boldsymbol{h}_1^{(t)} = \boldsymbol{c}_1^{(t)} \otimes tanh(\boldsymbol{x}^{(t)}\boldsymbol{W}^{(\boldsymbol{x},\boldsymbol{h})} + \boldsymbol{h}_N^{(t-1)}\boldsymbol{W}_1^{(\boldsymbol{h})}) + (1 - \boldsymbol{c}_1^{(t)}) \otimes \boldsymbol{h}_N^{(t-1)} \tag{2.2}$$

$$\boldsymbol{c}_\ell^{(t)} = sigmoid(\boldsymbol{h}_{j_\ell}^{(t)}\boldsymbol{W}_{\ell,j_\ell}^{(\boldsymbol{c})}) \tag{2.3}$$

$$\boldsymbol{h}_\ell^{(t)} = \boldsymbol{c}_\ell^{(t)} \otimes f_\ell(\boldsymbol{h}_{j_\ell}^{(t)}\boldsymbol{W}_{\ell,j_\ell}^{(\boldsymbol{h})}) + (1 - \boldsymbol{c}_\ell^{(t)}) \otimes \boldsymbol{h}_{j_\ell}^{t} \tag{2.4}$$

A state-of-the-art work of NAS called DARTs which used the same search space as mentioned before was proposed recently. By assigning all possible components of a RNN model with continuous weights (called architecture parameters in this paper), it could update the parameters in linear layers and architecture together by gradient descent. However, when it was applied to the MTSP tasks, the constraints in the search space limited its ability

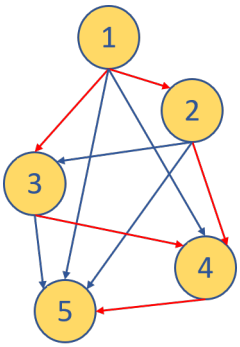


Figure 2.1: DAG representation of the RNN cell: To avoid the cycles in the graph, each node is assigned an index and only accepts the input from the nodes with lower indexes. In this example, there are 10 possible connections among nodes in total, colored as blue or red. The set of arrows in red color is one possible combination of connections among the nodes.

to adapt to the data with different properties.

Chapter 3

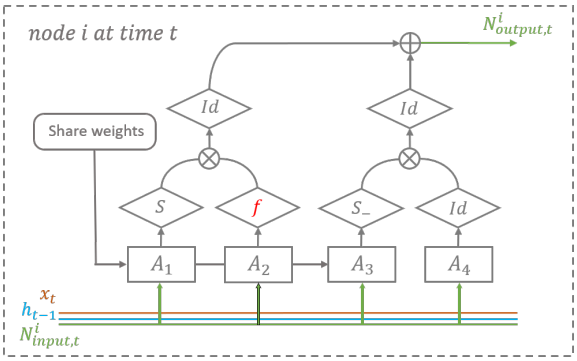
Proposed Approach

This chapter presents PS-DARTs in details: Section 3.1 demonstrates how we extend the search space for each node in DARTs. At the end of this section, we also describe how to generate discrete architectures from the search results. Section 3.2 shows how the progressively stacking framework conducts the search for each node one by one together with the connections of the whole graph.

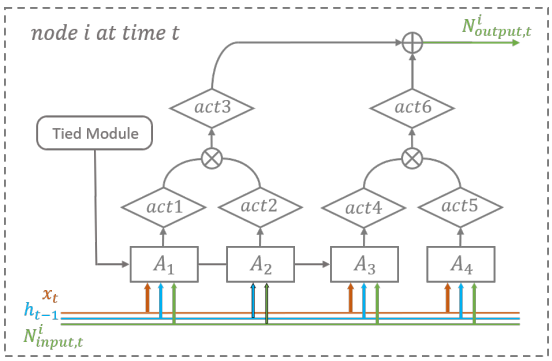
3.1 Search Space

In our work, we follow the representation of the connections inside the RNN cell as Fig. 2.1. For the microarchitecture inside each node, inspired by the learnable gating framework in vanilla RNN [24], long short-term memory model (LSTM) [14, 25], peephole LSTM [12, 26], gated recurrent unit (GRU) [27] and the variants of LSTM/GRU listed in [28], we formulate the search space of each node as a combination of different gates and relax the constraints of RHN backbone. As shown in the Fig. 3.1(b), each node consists of 6 activation selection modules (ASM), 4 addition modules (AM) and 1 tied module (TM), where the ASMs and AMs are used to search appropriate activation operations and latent representations, respectively. TM decides whether to share the input between AM1 and AM3 like RHN. All modules are embedded in the skeleton with 2 multiplication and 1 addition operation, which remain constant and are not part of the search space.

To learn how the combination of different computational components affects the model's performance through back propagation, each computational component is assigned a weight, which is also called architecture parameters.



(a) Search space in previous work: For the node with index i and $i > 1$, there is only the module labeled as f can attend into search. For the input of ASM A_1-A_4, only the output from previous nodes is considered. Note, x_t and h_{t-1} are only fed into the first node which is fixed to RHN and will not attend into the search.



(b) Search space updated: Compared with the diagram on the left, we extend each part where we can add activation functions and the decision of whether to tie inputs between A_1 and A_3 to be searchable. Besides, every node can consider taking x_t and h_{t-1} as possible inputs regardless of the index.

Figure 3.1: Search space in a single node i at time step t: the modules labeled as 'A' are addition modules (AM); the modules labeled as 'act' are activation selection modules (ASM); the arrows pointing to each AM are possible latent representations for AM's input, x_t is the input at time t; h_{t-1} represents the hidden states from last time step $t-1$; $N^i_{input,t}$ is the output from previous node for node i at time t; $N^i_{output,t}$ is the output for node i at time t. To distinguish the input of A_2 from other ASMs, we add black outlines to the arrows which mean no linear layer will be added to h_{t-1} or $N^i_{input,t}$.

3.1.1 Architecture Parameters

Architecture parameters consist of five parts: ASM architecture parameters $\boldsymbol{\alpha}_{ASM}$, AM architecture parameters $\boldsymbol{\alpha}_{AM}$, tied architecture parameters $\boldsymbol{\alpha}_{TM}$, and connection architecture parameters $\boldsymbol{\alpha}_{N}$, whose details and their corresponding modules will be introduced in the following sections.

3.1.1.1 Activation Selection Module (ASM)

There are four activation functions in each ASM: *sigmoid*, *Relu*, *tanh*, and *identity*, noted as $S, R, T, Id, S\_$, respectively. Each of them is assigned a weight $(\boldsymbol{\alpha}_{ASM})_j^o$, where $o \in S, R, T, Id, S\_$, j is the index of *ASM* module, $1 \leq j \leq 6$. Eq. 3.1 shows how the ASM normalizes $(\boldsymbol{\alpha}_{ASM})_j$ based on the input:

$$ASM_j(\boldsymbol{x}) = \sum_o ReLu(softmax((\boldsymbol{\alpha}_{ASM})_j)^o - C_{ASM}) K_{ASM} * f_o(\boldsymbol{x}) \tag{3.1}$$

, where $f_S(x) = Sigmoid(x)$, $f_R(x) = LeakyRelu(x)$, $f_T(x) = Tanh(x)$, $f_{Id}(x) = x$, $f_{S\_}(x) = 1 - Sigmoid(x)$, C_{ASM} and K_{ASM} are constant coefficients. Because the architecture is the same across different time steps and each node is optimized one by one, we ingore the time step and the node index in the definition of the variables for all architecture parameters assuming the current time step is t and the node index is i unless explicitly stated.

ReLU activation after shifting operation enables us to update the unwanted weights to 0 or close to 0. Thus, the result of the the architecture parameters is close to discrete representation, which can reduce degradation brought by discretizing step.

3.1.1.2 Addition Module (AM)

Each possible latent representation in AM will be assigned a weight $(\boldsymbol{\alpha}_{AM})_j^k$, where j is the index of *AM* module, $1 \leq j \leq 4$; k is the index of latent representations, $1 \leq k \leq l_j$; l_j is the number of all possible latent representations. Eq. 3.2 shows how AM normalizes

$\boldsymbol{\alpha}_{AM}$ based on the input:

$$AM_j(\boldsymbol{L}_j) = \sum_k^{l_j} ReLU(softmax((\boldsymbol{\alpha}_{AM})_j)^k - C_{AM})K_{AM} * \boldsymbol{L}_j^k \tag{3.2}$$

, where C_{AM} and K_{AM} are constant coefficients. As shown in Fig. 3.1(b), for $j \in \{1,3,4\}$: $l_j = 4$, $\boldsymbol{L}_j = [\boldsymbol{W}_j^{\boldsymbol{x}}\boldsymbol{x}_t, \boldsymbol{W}_j^{\boldsymbol{h}}\boldsymbol{h}_{t-1}, \boldsymbol{W}_j^{\boldsymbol{N}}\boldsymbol{N}_t^i, \boldsymbol{W}_b]$, where $\boldsymbol{x}_t$ is the input x at time step t; $\boldsymbol{h}_{t-1}$ is the hidden state at time step $t-1$; $\boldsymbol{N}_t^i$ is the input of node i at time step t; $\boldsymbol{W}_b$ is bias; $\boldsymbol{W}_j^{\boldsymbol{x}}$, $\boldsymbol{W}_j^{\boldsymbol{h}}$ and $\boldsymbol{W}_j^{\boldsymbol{N}}$ are parameters of the linear layers. For $j = 2$: $l_j = 2$, $\boldsymbol{L}_j = [\boldsymbol{h}_{t-1}, \boldsymbol{N}_t^i]$.

3.1.1.3 Tied Module (TM)

To cover more architectures, tied architecture weights $\boldsymbol{\alpha}_T$ is introduced to decide whether to share weights between AM_1 and AM_3. Eq. 3.3 shows how $\boldsymbol{\alpha}_T$ is normalized:

$$TM(\boldsymbol{x}) = (\boldsymbol{\alpha}'_{TM})^1 * (1 - AM_1(\boldsymbol{x})) + (\boldsymbol{\alpha}'_{TM})^2 * AM_3(\boldsymbol{x}) \tag{3.3}$$

$$\boldsymbol{\alpha}'_{TM} = ReLU(softmax(\boldsymbol{\alpha}_{TM}) - C_T)K_T \tag{3.4}$$

, where $\boldsymbol{\alpha}'_{TM} = [(\boldsymbol{\alpha}'_{TM})^1, (\boldsymbol{\alpha}'_{TM})^2]$, C_T and K_T are constant coefficients.

We also add a loss function here to avoid that two options have equal weights, in which case $TM(\boldsymbol{x})$ keeps a constant value.

$$\mathcal{L}_T = -((\boldsymbol{\alpha}'_{TM})^1 - (\boldsymbol{\alpha}'_{TM})^2) * ((\boldsymbol{\alpha}'_{TM})^1 - (\boldsymbol{\alpha}'_{TM})^2) \tag{3.5}$$

3.1.1.4 Connections Among Nodes

Similar to the work in DARTs [17], the nodes are sorted by their indexes and the connections can only start from the nodes with lower indexes to the ones with higher indexes. Fig. 3.2 shows an example of search space among nodes, where $e_{(n_1,n_2)}$ represents the edge from node n_1 to node n_2 and it will be assigned an weight $(\boldsymbol{\alpha}_N)_{n_2}^{n_1}$ during search. Eq. 3.6

shows how $(\boldsymbol{\alpha}_N)_{n_2}$ is normalized and the input for node n_2 is calculated:

$$\boldsymbol{N}_{input}^{n_2} = \sum_{n_1=1}^{n_1<n_2} ReLU(softmax((\boldsymbol{\alpha}_N)_{n_2})^{n_1} - C_N)K_N * \boldsymbol{N}_{output}^{n_1} \tag{3.6}$$

, where C_N and K_N are constant coefficients. For the output of the whole RNN cell, we consider it as a virtual node (consist of only one identity function) has searchable connections with all nodes.

With all the equations mentioned before and Fig. 3.1(b), the output for node n_1 can be formulated as following:

$$N_{output}^{n_1} = ASM_3^{n_1}(ASM_1^{n_1}(AM_1^{n_1}(\boldsymbol{L}_1^{n_1})) * ASM_2^{n_1}(AM_2^{n_1}(\boldsymbol{L}_2^{n_1}))) \tag{3.7}$$

$$+ASM_6^{n_1}(ASM_4^{n_1}(TM^{n_1}(AM_1^{n_1}(\boldsymbol{L}_1^{n_1}), AM_3^{n_1}(\boldsymbol{L}_3^{n_1})) * ASM_5^{n_1}(AM_4^{n_1}(\boldsymbol{L}_4^{n_1})))) \tag{3.8}$$

With this design, the complexity of ASM is 4^6, AM is $2^{(3\times3+2)}$, TM is 2 per node, connections among nodes is $N!$. Thus, the complexity of whole search space is $(4^6 \times 2^{(3\times3+2)} \times 2)^N \times N! \approx 2e^{95}$ when $N = 12$, which is $2e^{85}$ larger than the one in DARTs.

3.1.2 Initializing Architecture Parameters and Deriving Discrete Architectures

In PS-DARTs, the architecture of each node will be searched one by one, thus the initialization and discretization of architecture parameters for each node are also executed one by one. In the initialization step, all parameters in each module will be assigned the same value. The discretization works as follows:

- ASM architecture parameters: Only the activation function with the maximum weight in each module will be kept, all others will be deleted as [17] to reduce the model's complexity.
- AM architecture parameters: All latent components with normalized weights $\boldsymbol{\alpha}_{AM}$ larger than 0 will be kept, the other components will be deleted.

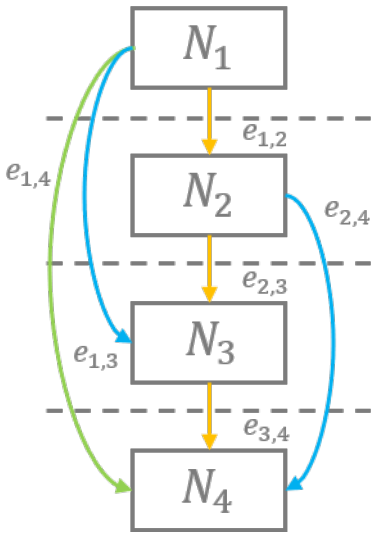


Figure 3.2: Progressive search: First, add node N_1 to the graph and search the architecture of it. Second, fix architecture of discretized N_1, add node N_2 and edge $e_{1,2}$ to the graph and search the micro architecture in node N_2. Third, fix architecture of discretized N_3, add node N_3, edge $e_{1,3}$ and $e_{2,3}$ to the graph and search the micro architecture in node N_3.

- TM architecture parameters: The decision with the larger weight α_{TM} will be kept, the other one will be deleted.
- Connection architecture parameters: Only connections with normalized weight α_N larger than 0 will be kept, all other connections will be deleted.

Meanwhile, only after searching for all nodes is finished, the architecture weights for connections α_N among nodes is discretized. Before that, every time a new node is added to the search space, the architecture parameters for all the connections among nodes will be reset to the initial state.

3.2 Progressively Stacking Search Algorithm

If we consider all nodes as part of the search space at the same time, memory space and searching time increases drastically. To solve this problem, we design a greedy progressive approach as shown in Alg. 1 to study the connections among the nodes and the architecture of each node at the same time.

Algorithm 1 Progressively Stacking Differentiable Architecture Search

1: **input:** graph G, architecture parameter α, linear weights ω
2: **while** not exceeds parameter budget **or** not increase in performance **do**
3: Add new node to G
4: Pretrain ω in G
5: **while** not converged **do**
6: Update α by descending $\nabla \mathcal{L}_{val}(\omega - \eta \nabla_{\omega} \mathcal{L}_{train}(\omega, \alpha), \alpha)$
7: Update weights ω by descending $\nabla \mathcal{L}_{train}(\omega, \alpha)$;
8: **end while**
9: Discretize current node
10: **end while**

3.2.1 Definitions in Graph

Graph G includes nodes and connections among nodes as shown in Fig. 2.1. Each node consists of the parameters in linear layers (which are also called parameters in the following sections) ω' and architecture parameters α'. The collection of parameters for all nodes in G is called ω, and the collection of architecture parameters for all nodes and connections is called α.

3.2.2 Discrete Current Node

Once the architecture of the currently searching node is discretized following section 3.1.2, all parameters of a current node are reset to the initial state and the kept architecture parameters are fixed in the following steps. This process corresponds to line 9 in Alg. 1.

3.2.3 Add New Node

Then, a new node and the connections pointed from previous nodes to this new node are added to G. Besides, the parameters and architecture parameters of the new node are added to ω and α, respectively. This process corresponds to line 3 in Alg. 1.

3.2.4 Pretrain Parameters

Unlike only selecting activation functions, when we add the linear components to the search space, the random initialization of parameters in linear layers may affect the decision on architecture. Thus, we add a pretrain step before updating architecture parameters. The exact number of this pretrain step will be decided by grid search described in section 4.4.1.1. This process corresponds to line 4 in Alg. 1.

3.2.5 Update Parameters and Architecture Parameters

After pretraining all parameters in G, the architecture parameters for this new node as well as the connections among all nodes will attend in the search. Then, all the linear parameters and architecture parameters in the graph will be updated by gradient descent alternately in each epoch until converging as [17]. Repeat this process until adding a new node exceeds the parameter budget or can not improve the performance further. Here the parameter budget refers to the maximum parameter number which does not exceed the GPU memory, which is never reached in our experiment. This process corresponds to line 5-8 in Alg. 1.

The loss function $\mathcal{L}$ here is the summation of $\mathcal{L}_T$ absolute loss (L1-loss) and as shown in Eq. 3.9:

$$\mathcal{L} = \mathcal{L}_T + \sum_{\Omega} \sum_{t} \sum_{i=0}^{n-1} |Y_{t,i} - \hat{Y}_{t,i}| \tag{3.9}$$

, where n is the number of variables, t is time step, Ω is dataset which can be either training set or validation set and $Y, \hat{Y} \in \mathbb{R}^{n \times t}$ are ground true data and prediction data, respectively.

Chapter 4

Experiments

In this section, we compare our approach with 9 state-of-the-art approaches on 3 benchmark datasets for MTSP tasks.

4.1 Datasets

We use following three benchmark datasets that are publicly available:

- **Traffic**[1]: Hourly road occupancy rates (between 0 and 1) collected by different sensors on San Francisco Bay area freeways from 2015 to 2016 (48 months).
- **Electricity**[2]: The electricity consumption in the unit of kWh collected every 15 minutes from 2012 to 2014 for 321 clients. In our experiment, it is converted into hourly data.
- **Exchange-Rate**: Daily exchange rate collected from eight countries including Australia, British, Canada, Switzerland, China, Japan, New Zealand and Singapore from 1990 to 2016

We follow the pre-processing step as [10] where each dataset is split into the training set, validation set and test set by 6:2:2 in chronological order.

In LST-skip/attn's work, the electricity and traffic datasets are proved to have both short-term daily patterns and long-term weekly patterns. While there is no repetitive long-term patterns in exchange rate dataset. These observations are important for our later analysis on the comparisons of different approaches. The manually designed complicated models perform better on predicting the data with obvious repetitive patterns than the one without

[1] http://pems.dot.ca.gov

[2] https://archive.ics.uci.edu/ml/datasets/ElectricityLoadDiagrams20112014

repetitive patterns. Since the proposing of the NAS is to automatically design an appropriate architecture for a specific task, a good NAS framework should be able to design the models adaptive to the data with various repetitive patterns. In the section 4.5 we will do an empirical analysis for this point.

4.2 Methods for Comparison

In our work, we compare following methods:

1. AR: One dimension Auto Regressive model.
2. LRidge: Vector Auto Regression (VAR) model with L2-regularization.
3. LSVR [29]: VAR model with Support Vector Regression (SVR) objective function.
4. GP [30]: Gaussian Process for time series modeling.
5. RNN-GRU [7]: Recurrent Neural Network using Gate Recurrent Unit.
6. LST-skip [10]: Long-Short-Term Temporal model with skip RNN layers.
7. LST-atten [10]: Long-Short-Term Temporal model with temporal attention layers.
8. TPA-LSTM [11]: Long-Short-Term Memory (LSTM) with Temporal Attention (TPA).
9. DARTs [17]: The first work of Differentiable Architecture Search.
10. PS-DARTs: Our proposed Progressively Stacking Differentiable Architecture Search.

Among these methods: 1-8 are manually designed models. There is only one architecture for each method on three datasets with only hyperparameters being different. 9-10 are NAS approaches that generate different RNN models for different datasets.

4.3 Metrics

We use the following three evaluation metric to compare the performances of different models:

- **Relative Squared Error (RSE)**:

$$RSE = \frac{\sqrt{\sum_{(i,t)\in\Omega_{test}} (Y_{it} - \hat{Y}_{it})^2}}{\sqrt{\sum_{(i,t)\in\Omega_{test}} (Y_{it} - mean(Y))^2}} \tag{4.1}$$

- **Correlation Coefficient** (**CORR**):

$$CORR = \frac{1}{n}\sum_{i=1}^{n} \frac{\sum_t (Y_{it} - mean(Y_i))(\hat{Y}_{it} - mean(\hat{Y}_i))}{\sum_t \sqrt{(Y_{it} - mean(Y_i))^2 (\hat{Y}_{it} - mean(\hat{Y}_i))^2}} \tag{4.2}$$

- **Searching Efficiency (SE)** :

$$SE = \frac{number\ of\ architecture\ parameters}{searching\ time} \tag{4.3}$$

, where t is time step; n si the length of time series data; i is index of variables; Ω_{test} is test dataset; $Y, \hat{Y} \in \mathbb{R}^{n\times t}$ are ground true data and prediction data, respectively; searching time here is in the unit of GPU hour. The RSE is the scaled version of Root Mean Square Error (RMSE), which is designed to eliminate the effects of data scale. For RSE, the lower the better, whereas for CORR and SE, the higher the better.

4.4 Experiment Details

4.4.1 Hyperparameters

We used the source code in of LST-skip [10] and replaced its deep learning model with our PS-DARTs framework. However, introducing NAS adds two new hyperparameters:

architecture learning rate: lr_{arch} and pretrain epochs: $epoch_t$. For these hyperparameters we conducted a grid search. For the other hyperparameters, we followed the setting up in TPA-LSTM whose model is more similar to ours. Thus, the settings for hyperparameters are divided into the following two parts.

4.4.1.1 Hyperparameter Search

- lr_{arch}: Since architecture parameters and parameters in linear layers are updated alternatively, they should have separate learning rates. We followed the setting up of lr as TPA-LSTM, then we set up the grid search range based on lr: $\{0.01 * lr, 0.05 * lr, 0.1 * lr, 0.5 * lr, lr, 5 * lr, 10 * lr\}$.

- $epoch_t$: There is no pretrain step in DARTs and we want to explore how many epochs of pretraining enable the parameters in linear layers to neither affect the decision of component selection nor overfit the model. We set up the grid search range as $\{0, 5, 10, 15\}$.

4.4.1.2 Constraints on Hyperparameters

NAS focused on finding architectures for each problem, we can assume that the searching algorithm should be able to find an appropriate architecture given a set of specific hyperparameters related to the model's structure. To save time on the hyperparameter searching, we select the hyperparameters for NAS as follows:

- **Learning Rate**: Same as TPA-LSTM, where the learning rate is 0.001 for electricity and traffic datasets and 0.003 for exchange rate dataset.

- **Hidden Size**: The maximum hidden size in TPA-LSTM's hyperparameter search space, where the hidden size is 70 for electricity and traffic datasets and 12 for exchange rate dataset.

- **Window Size**: The maximum size in TPA-LSTM's hyperparameter search space which doesn't exceed the computer's RAM or searching time threshold(1 minute) for a single epoch.

- **Dropout**: The maximum value in TPA-LSTM's hyperparameter search space, which is 0.2 for all datasets.

- **Searching Epochs**: We considered the searching to be converged if the discrete architecture derived from the current epoch hasn't been changed for 15 epochs and the searching will be stopped.

- **Training Epochs**: We considered the training to be converged if no validation RSE loss decreases or validation CORR increases for 50 epochs and the training will be stopped.

- **Other Hyperparameters**: Batch size, auto regression window size, optimizer, weight decay are not mentioned in other works, thus we followed the settings in LST-skip's source code.

4.4.2 Experiment Devices

The experiments of DARTs and PS-DARTs are conducted on the NVIDIA Tesla V100 GPU with Pytorch framework and TorchScript accelerator.

4.5 Results and Analysis

We test PS-DARTs on the three MTS datasets mentioned above with the evaluation matrix of RSE and CORR. Then we compare it with the experiment results of traditional approaches, LSTNet-skip, LSTN-attn and TPA-LSTM. Besides, we also reimplement DARTs on these three MTS datasets and compare its SE, RSE and CORR with PS-DARTs. Finally, our approach achieves 16 best result and 6 second best on 24 tasks.

Dataset		Exchange-rate				Traffic				Electricity			
		Horizon				Horizon				Horizon			
Methods	Metric	3	6	12	24	3	6	12	24	3	6	12	24
AR	RSE	0.0228	0.0279	0.0353	0.0445	0.5991	0.6218	0.6252	0.6293	0.0995	0.1035	0.1050	0.1054
	CORR	0.9734	0.9656	0.9526	0.5357	0.7752	0.7568	0.7544	0.7519	0.8845	0.8632	0.8591	0.8595
LRidge	RSE	0.0184	0.0274	0.0419	0.0675	0.5833	0.5920	0.6148	0.6025	0.1467	0.1419	0.2129	0.1280
	CORR	0.9788	0.9722	0.9543	0.9305	0.8038	0.8051	0.7879	0.7862	0.8890	0.8594	0.8003	0.8806
LSVR	RSE	0.0189	0.0284	0.0425	0.0662	0.5740	0.6580	0.7714	0.5909	0.1523	0.1372	0.1333	0.1180
	CORR	0.9782	0.9697	0.9546	0.9370	0.7993	0.7267	0.6711	0.7850	0.8888	0.8861	0.8961	0.8891
GP	RSE	0.0239	0.0272	0.0394	0.0580	0.6082	0.6772	0.6406	0.5995	0.1500	0.1907	0.1621	0.1273
	CORR	0.8713	0.8193	0.8484	0.8278	0.7831	0.7406	0.7671	0.7909	0.8670	0.8334	0.8394	0.8818
GRU	RSE	0.0192	0.0264	0.0408	0.0626	0.5358	0.5522	0.5562	0.5633	0.1102	0.1144	0.1183	0.1295
	CORR	0.9786	0.9712	0.9531	0.9223	0.8511	0.8405	0.8345	0.8300	0.8597	0.8623	0.8472	0.8651
LSTNet-skip	RSE	0.0226	0.0280	0.0356	0.0449	0.4777	0.4893	0.4950	0.4973	0.0864	0.0931	0.1007	0.1007
	CORR	0.9735	0.9658	0.9511	0.9354	0.8721	0.8690	0.8614	0.8588	0.9283	0.9135	0.9077	0.9119
LSTNet-attn	RSE	0.0276	0.0321	0.0448	0.0590	0.4897	0.4973	0.5173	0.5300	0.0868	0.0953	0.0984	0.1059
	CORR	0.9717	0.9656	0.9499	0.9339	0.8704	0.8669	0.8540	0.8429	0.9243	0.9095	0.9030	0.9025
TPA-LSTM	RSE	0.0174	0.0241	0.0341	0.0444	0.4487	0.4658	0.4641	0.4765	0.0823	0.0916	0.0964	0.1006
	CORR	0.9790	0.9709	0.9564	0.9381	0.8812	0.8717	0.8717	0.8629	0.9429	0.9337	0.9250	0.9133
DARTs*	RSE	0.0209	0.0273	0.0347	0.0458	0.4583	0.4663	0.4736	0.4991	0.0836	0.0944	0.0978	0.0979
		±0.0008	±0.0006	±0.0004	±0.0007	±0.0014	±0.0010	±0.0007	±0.0101	±0.0004	±0.0007	±0.0004	±0.0006
	CORR	0.9961	0.9941	0.9921	0.9862	0.8789	0.8745	0.8713	0.8548	0.9365	0.9260	0.9242	0.9203
		±0.0002	±0.0002	±0.0014	±0.0005	±0.0006	±0.0009	±0.0010	±0.0018	±0.0013	±0.0003	±0.0009	±0.0007
PS-DARTs	RSE	0.0174	0.0244	0.0332	0.0436	0.4544	0.4645	0.4704	0.4843	0.0834	0.0916	0.0947	0.0964
		±0.0002	±0.0003	±0.0002	±0.0007	±0.0009	±0.0017	±0.0013	±0.0024	±0.0007	±0.0016	±0.0013	±0.0005
	CORR	0.9968	0.9950	0.9917	0.9868	0.8823	0.8765	0.8727	0.8661	0.9349	0.9280	0.9239	0.9205
		±0.0000	±0.0001	±0.0002	±0.0001	±0.0006	±0.0007	±0.0009	±0.0006	±0.0026	±0.0021	±0.0019	±0.0011

Table 4.1: Comparison with state-of-the-art models on three time series datasets: Red means the best among all approaches. Green means the second best. We repeat the experiments on each task for 5 times and report the mean/standard deviation information. * means we reimplement DARTs on MTS datasets.

4.5.1 Comparison with Traditional Approaches & Deep Learning Models

The result of each approach is shown in table 4.1, where the experiment result of AR, LRidge, LSVR, GP, GRU, LSTNew-skip/attn are from [10], the experiment result of TPA-LSTM is from [11] and DARTs is reimplemented on the three MTS datasets.

Although TPA-LSTM exceeds the performance of traditional approaches on traffic and electricity datasets, it performs slightly worse than the traditional approaches when horizon is 6 on exchange rate dataset. This shows the limitation of TPA-LSTM that can not adaptive to various MTS datasets with different repetitive patterns. PS-DARTs exceeds all the traditional approaches on all tasks, which in turn proves that our approach can design appropriate structures for tasks with various repetitive patterns. For the comparison with TPA-LSTM on traffic and electricity datasets, PS-DARTs performs slightly worse than but still competitive as TPA-LSTM in term of RSE with horizon 1, 12, 24 on traffic dataset and CORR with horizon 3, 6, 12 on electricity dataset while achieves the best performance on

all the other 9 tasks. However, neither does our approach conduct any exhaustive hyperparameter search nor adds any extra attention modules like TPA-LSTM or LSTNet-skip/attn. This further shows PS-DARTs' power of automatically finding appropriate architectures for specific tasks.

4.5.2 Comparison with DARTs

When apply NAS on three MTS datasets, we do hyperparameter search for DARTs on architecture parameters and PS-DARTs on both architecture parameters and pretraining epochs in the searching period. Unlike PS-DARTs, the number of RNN layers in DARTs can only be decided manually before searching but not automatically during searching. To decided this hyperparameter, we first select the best architecture found by PS-DARTs. Then we define the number of RNN layers by keeping the number of trainable parameters the same as DARTs. All other hyperparameters follow the settings in section 4.4.1.2.

Methods	Dataset	Exchange-rate	Traffic	Electricity
DARTs	GPU hours	0.06	0.33	0.45
	Arch Params	4	10	17
	Efficiency	**66.67**	30	37.78
PS-DARTs	GPU hours	0.98	2.61	2.75
	Arch Params	53	108	164
	Efficiency	54.08	**41.37**	**59.64**

Table 4.2: Comparison of DARTs and PS-DARTs on searching efficiency

The comparison between DARTs and PS-DARTs' searching efficiency of the best architecture found on each task is shown in table 4.2. On each task, the number of architecture parameters in PS-DARTs is around 10 times of that in DARTs, which means more possible architectures are explored. Although the searching time is increased, PS-DARTs in fact improves the searching efficiency on traffic and electricity datasets. On the exchange-rate dataset, DARTs' efficiency is around 1.2 of PS-DARTs, which is caused by PS-DARTs' searching time overhead. DARTs fixes the first layer to RHN and only conducts search for the other layers. In contrast, PS-DARTs searches the architecture for each layer and needs

to search one layer further to decide at which step to stop the search. This overhead affects the searching efficiency when the number of layer is small. In our reported result, the best architecture found by PS-DARTs has only one layer. Thus, the searching efficiency is slightly worse than DARTs.

For the comparison of RSE and CORR, PS-DARTs exceeds DARTs on 22 tasks. For the other two tasks, PS-DARTs still shows competitive performances as DARTs. This then demonstrates that the expanding of the search space in PS-DARTs can lead us to find better architectures that can capture more precise dynamics of the MTS with different repetitive patterns.

Chapter 5

Conclusion

In this work, we propose PS-DARTs which can find better architectures in a competitive searching time as DARTs. Our experiments on three real-world MTS datasets strongly support this idea and show PS-DARTs has better searching efficiency than DARTs and ability to explore more accurate architectures for MTS data with different repetitive patterns than manually designed deep learning models and DARTs.

BIBLIOGRAPHY

[1] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[2] William WS Wei. Time series analysis. In *The Oxford Handbook of Quantitative Methods in Psychology: Vol. 2*. 2006.

[3] Jiahan Li and Weiye Chen. Forecasting macroeconomic time series: Lasso-based approaches and their forecast combinations with dynamic factor models. *International Journal of Forecasting*, 30(4):996–1015, 2014.

[4] Hsiang-Fu Yu, Nikhil Rao, and Inderjit S Dhillon. Temporal regularized matrix factorization for high-dimensional time series prediction. *Advances in neural information processing systems*, 29:847–855, 2016.

[5] G Peter Zhang. Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175, 2003.

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

[7] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[9] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[10] Guokun Lai, Wei-Cheng Chang, Yiming Yang, and Hanxiao Liu. Modeling long-and short-term temporal patterns with deep neural networks. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 95–104, 2018.

[11] Shun-Yao Shih, Fan-Keng Sun, and Hung-yi Lee. Temporal pattern attention for multivariate time series forecasting. *Machine Learning*, 108(8):1421–1441, 2019.

[12] Felix A Gers and E Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[13] Ben Krause, Liang Lu, Iain Murray, and Steve Renals. Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

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

[15] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[16] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384, 1989.

[17] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[18] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnık, and Jürgen Schmidhuber. Recurrent highway networks. In *International Conference on Machine Learning*, pages 4189–4198. PMLR, 2017.

[19] Li-Juan Cao and Francis Eng Hock Tay. Support vector machine with adaptive parameters in financial time series forecasting. *IEEE Transactions on neural networks*, 14(6):1506–1518, 2003.

[20] John R Freeman, John T Williams, and Tse-min Lin. Vector autoregression and the study of politics. *American Journal of Political Science*, pages 842–877, 1989.

[21] Roger Frigola and Carl Edward Rasmussen. Integrated pre-processing for bayesian nonlinear system identification with gaussian processes. In *52nd IEEE Conference on Decision and Control*, pages 5371–5376. IEEE, 2013.

[22] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR, 2018.

[23] Xiangning Chen and Cho-Jui Hsieh. Stabilizing differentiable architecture search via perturbation-based regularization. In *International Conference on Machine Learning*, pages 1554–1565. PMLR, 2020.

[24] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[25] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[26] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

[27] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[28] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[29] Vladimir Vapnik, Steven E Golowich, Alex Smola, et al. Support vector method for function approximation, regression estimation, and signal processing. *Advances in neural information processing systems*, pages 281–287, 1997.

[30] Roger Frigola. *Bayesian time series learning with Gaussian processes*. PhD thesis, University of Cambridge, 2015.