Distributed Learning in Sensor Networks

— an online-trained spiral recurrent neural network, guided by an evolution framework, making duty-cycle reduction more robust

Huaien Gao

München 2008

Distributed Learning in Sensor Networks

— an online-trained spiral recurrent neural network, guided by an evolution framework, making duty-cycle reduction more robust

Huaien Gao

Dissertation
an dem Institut für Informatik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Huaien Gao
geb. 25.10.1977

München, den 27.05.2008

Erstgutachter: Hans-Peter Kriegel
Zweitgutachter: Darius Burschka
Tag der mündlichen Prüfung: 26.Jan.2009

Acknowledgement

This thesis is completed within the joint Ph.D. program between the department of Database and Information Systems in the Institute for Computer Science in the University of Munich and the department of Learning System in Corporate Technology, Siemens AG. Many people have given valuable advises during the research and the writing. Without their support, this thesis cannot have been written.

Many thanks are due to Professor Hans-Peter Kriegel who has been supportive not only on my research activities but also on my application of "Aufenthaltsbewilligung" which is very important to me. Also thanks are due to Prof. Dr. Darius Burschka for his kindly agreement to act as the second examiner of this thesis.

I would like to express my sincere gratitude to Dr. Rudolf Sollacher, in Corporate Technology in Siemens AG, for he has guided, instructed, encourage, inspired and continually motivated me.

I am also very grateful to Dr. Paul-Theo Pilgram, in Corporate Technology in Siemens AG, for his generous hospitality to proofread the whole manuscript, and making the thesis much smoother.

I am indebted to Prof. Dr. Bernd Schürmann and Dr. Thomas Runkler, heads of department of Learning System in Corporate Technology of Siemens AG, who have been constantly supportive to my research and given me advises on the thesis.

Prof. Dr. Martin Greiner and Dr. Jochen Cleve are always helpful to me, no matter it concerns about my research or other difficulty.

I will remember the help from colleagues both in Siemens AG and in University of Munich. The following list is undoubtedly incomplete: Dr. Kai Yu, Anton Maximilian Schaffer, Dr. Kai Heesche, Dr. Christoph Tietz, Dr. Hans-Georg Zimmermann, Dr. Peter Mayer, Dr. Peter Kunath, Mrs. Susanne Grienberger, Dr. Volker Tresp, Yi Huang, Dr. Marco Pellegrino, Mrs. Christa Singer.

More than to anyone else, I own to the constantly love and support from my family. In every stage of my life, they always encourage, support and understand me. No matter the difficulties I am facing, they are the ones who tell me never give up and give me the power to confront. This thesis is dedicated to them.

Abstract

The sensor networks of today tend to be built from “intelligent” sensor nodes. Such nodes have substantial processing capability and memory, are locally distributed and communicate wirelessly with one another. They are mostly battery-powered, possibly even with a lifetime battery. Power management is hence a prime concern.

Such intelligent sensors, or “smart sensors”, are typically involved in some diagnosis task, which would benefit greatly from an ability to predict environment data. The best of those predictions are obtained by training a learning-model with environment data. This task is non-trivial, computationally intensive and thus expensive on energy, particularly if the model imposed by the environment data is dynamic and complex. As the training data usually come from diverse sources, not only from the nearest sensor, the learning node must communicate with other nodes to get at their measurement data. Data processors can be made very energy efficient, whereas radio is inherently wasteful. The ultimate aim is to find the right balance between prediction quality and energy consumption.

Unlike conventional energy management solutions, which provide routing methods or communication protocols, this thesis introduces an efficient learning algorithm which improves prediction performance of sensors and reduces computational complexity. It introduces two techniques which both reduce the overall energy consumption of the sensor network: intra-node and inter-node solutions.

Intra-node solution: A sensor’s duty cycle is the time fraction during which the sensor is active. Battery life can be extended by reducing the duty cycle of the sensor. Depending on the communication protocol, radio communication coincides with more or less of the sensor’s active time. The radio duty cycle can be reduced by communicating less frequently. This thesis introduces Spiral Recurrent Neural Networks (*SpiralRNN*), a novel model for on-line prediction, where some received data get substituted by predicted data. It is shown that the *SpiralRNN* model works reliably, thus opening a way to significant energy savings.

Inter-node solution: Communication between sensor nodes can also be diminished at network level. Transmitting data (and receiving them) consumes energy and blocks the airwaves, but the information transmitted is often irrelevant (depending on the application). This thesis introduces a heuristic evolutionary method, the *evolution* *framework*, which weighs up energy consumption against prediction performance, adapting its model structure to the environment data as well as to application constraints. The complexity of the model gets lowered by removing some hidden nodes. The communication effort gets reduced by removing dependencies on various "unimportant" data (which makes communication dispensable in those cases).

Spiral Recurrent Neural Networks (*SpiralRNN*), in combination with *duty-cycle reduction* and the *evolution framework*, are a powerful technique for balancing prediction performance against energy consumption, and are hence valuable in the construction of sensor network applications.

Contents

Chapter 1

Introduction

Recent decades have seen widespread deployment of embedded electronic devices with forever increasing computational capabilities [1, 2, 3, 4, 5]. Examples of such devices are mobile phones, personal digital assistants, but also wireless sensor nodes [6, 7, 8, 9, 10, 11, 12]. The latter in particular - besides measuring all kinds of quantities like temperature, pressure and luminosity - are able to communicate with neighboring sensor nodes and to exchange information with each other. Due to these features, many sensor nodes together constitute a sensor network, which is applied at a large scale in periodical environment monitoring [13, 14, 15], such as pollution detection, building management, production monitoring, traffic analysis, and temperature recording.

The conventional approach for sensor network applications consists in spreading sensors across the area of interest, making them measure the relevant data, and to have a central station which collects all the information from surrounding sensors and which processes the data thereafter. Even though energy for one single sensor could be saved by increasing the number of sensor nodes, so that communication via multi-hop transmission is cheaper in this high-density network, the trend [16] is to put more computing capability into the sensor node itself and to communicate with other nodes only when necessary, thus saving even more energy. Sensor nodes in the network are not subordinate to any other node but are able to collect, process and communicate information at their own will. Together they constitute a distributed sensor network.

1.1 Goal

Designing and setting up a distributed sensor network is a complicated engineering task, comprising issues [16] like: selection of the collaborative signal processing algorithms executed in each sensor node; selection of the multi-hop networking algorithms; optimal matching of sensor requirements with communication performance; designing a security protocol for this pervasive network.

This thesis focuses on designing, for sensor networks with a prediction service, an on-line learning algorithm for each sensor node, and an operating framework on the network level, with the overall aim of reducing energy consumption and adapting to shifts in the environment, while maintaining sensor network performance.

1.2 Challenges and Solutions

Challenges in designing such an on-line learning algorithm and operating framework will lie mainly in the following areas:

Autonomy and autarky

Sensor nodes are commonly deployed in locations where cable access is either impossible or expensive. Owing to this, most sensor nodes are battery powered and impossible, or expensive, to be recharged. Therefore, energy is extremely precious in sensor nodes, making energy management one of the paramount issues when designing sensor networks. Another consequence of autonomy is the need for sensor nodes to organize themselves by learning and adapting to a changing environment.

Limited computational power

Progress in integrated circuit technology has vastly improved the processing capacity (including the capability for data processing and data communication as well as storage memory) of sensor-node computers. Nonetheless, the limited available energy of sensor nodes implies the usage of microprocessors and memory components with reduced capabilities. Therefore, any information processing within the sensor nodes has to take into account the corresponding limitations. A straight-forward implementation of existing algorithms will fail in most cases.

Complex dynamics

The dynamics of the measured environment data can be complicated, for example if the current value of the dynamics depends on the long-term historical value, or if the dynamic model at one location is related to the one at another location. In some cases, therefore, the dynamics of the measured data are a complicated system with temporal and spatial characteristics.

The mentioned challenges in distributed sensor network applications are interrelated. To name a few: the limited energy can hinder a comprehensive evaluation, and can thus cause an early depletion of network energy; improvements in processing ability can optimize the consumption of energy; complicated dynamics will normally require more comprehensive data processing and larger storage memory; self-organization affects the demand for communication, which affects the consumption of energy.

The energy limitation is a major problem in these applications. Many studies have focused on this topic, particularly on the optimal topology of sensor networks [17, 18, 19]. This thesis shows a novel way of saving energy: by applying learning algorithms to the embedded sensor-node system. The approach is built around two central ideas: (1) putting intelligence into the sensor nodes, i.e. a prediction ability, which is essential for duty-cycle reduction, the reduction of communication activity; (2) using the *evolution framework*, i.e. giving sensor nodes the ability to exchange information and to adapt to the environment, and also to reduce the communication and computation effort to some extent.

1.2.1 Intelligence of Sensor Node

Requirements

In self-organizing autonomous distributed sensor networks, any solution for the prediction of sensor data must satisfy certain requirements. They will determine the design of the prediction model. They are:

$< \mathfrak{R}1 >$ Autonomy:

No or little *a priori* knowledge about the environment shall be available to the sensor nodes. This implies that there will be no pre-training of parameters for the given problems, and that the sensor nodes have to construct their prediction models basically from scratch and on-line.

$< \mathfrak{R}2 >$ Adaptivity:

Shifts in the environment, rarely known in advance, result in the switching of dynamics. Sensor nodes must be able to detect changes, and accordingly adjust the model parameters in real-time.

$< \mathfrak{R}3 >$ Efficiency:

As real-time reaction is required, the solution for prediction should be efficient in terms of fast convergence, computational costs (number of arithmetic operations) and data storage requirements. Candidate learning systems are required to be compact as well as robust.

$< \mathfrak{R}4 >$ Reliability:

Unreliability of one sensor node can dramatically degrade the performance of the whole network. Thus, reliability of the system is essential, particularly for long-run performance. The reliability requirement implies that instabilities and singularities should not occur during training.

Benefits

A focal "intelligent" feature is the sensor nodes' ability to predict future values of their own measurement data. Two use cases will illustrate the benefits:

1. Neighboring sensor nodes exchange their sensor data for resilience and for in-network data interpolation. In order to increase their lifetime, sensor nodes

have to reduce energy-consuming activities, in particular wireless communication with their neighbors. If the sensor nodes are able to predict their own sensor data and those of their neighbors, they can prolong the intervals between data exchanges, thus saving energy.

2. In control applications, sensor data may be required at regular time intervals. However, wireless communication is inherently unreliable: transmissions may sometimes be corrupted or may even fail, due to interference effects. If sensor nodes are able to predict sensor data, they can at least partially compensate for such faulty transmissions.

Task

Environment properties, such as temperature and smoke density, are the main objectives which sensor nodes measure and predict.

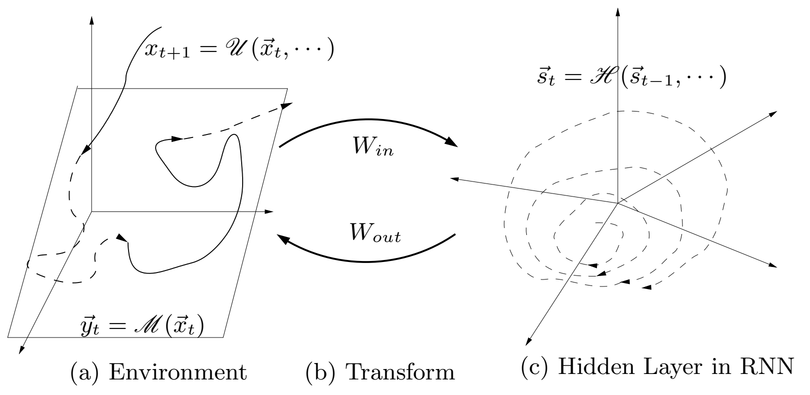
System dynamics of environment properties are normally temporal such that the current value depends on the previous status; it can also be spatial such that property values at one location depend on the data from their nearby surroundings. Therefore, using a description in discrete time-steps, numbered by variable t (same in the following chapters), one can use vector $\vec{x}_t$ to represent the property value at all the sensor nodes at time t, and use *eq-(1.1)* to represent the system dynamics of the environment in general, where symbol $\mathscr{U}$ denotes the system dynamics of the property in the domain. The measurement behaviour of the sensor node is expressed by *eq-(1.2)*. A sketch example of the environmental dynamics is shown in *fig-1.1(a)*. These properties in the domain satisfy certain physical dynamics equations which are usually nonlinear and complicated. However, the physical insight into the environment is a black box to the observer. One needs to find an approximation model describing the dynamics of the environment.

$$\vec{x}_{t+1} = \mathscr{U}(\vec{x}_t, \vec{x}_{t-1,} \cdots), \quad \text{where } \vec{x}_t = [x_{t,1}, x_{t,2}, \cdots]^T \tag{1.1}$$
$$\vec{y}_t = \mathscr{M}(\vec{x}_t) \tag{1.2}$$

Solution

The multi-layer perceptron (*MLP*) model is proven to be a universal approximator [20, 21] but it fails to model dynamics with long-term dependencies. Extensions of the *MLP* model, such as time-delayed neural networks (*TDNN*) [22, 23], can to some extent compensate for this drawback by assigning more data as input information, but this requires users to be aware of physical conditions which in many cases are not available (refer to $< \mathfrak{R}1 >$ on *page-3*).

Based on *MLP*, Recurrent Neural Network (RNN) models are able to approximate a large class of dynamic systems [24], including ones with long-term dependency, because RNN models possess recurrent neurons which can store history information from input data.



$x_{t+1} = \mathscr{U}(\vec{x}_t, \cdots)$
W_{in}
W_{out}
$\vec{s}_t = \mathscr{H}(\vec{s}_{t-1}, \cdots)$
$\vec{y}_t = \mathscr{M}(\vec{x}_t)$
(a) Environment
(b) Transform
(c) Hidden Layer in RNN

Figure 1.1: *Transforming between the dynamics of the environment and the ones of a recurrent neural network. (a) Environmental dynamics. The curve represents the dynamics $\mathscr{U}$ of the environment, which will be projected on the plane surface by the function $\mathscr{M}$ which reads out or measures the current data. (b) Transform of dynamics using multiplier matrices W_{in} and W_{out}. Matrix W_{in} imports the environmental data into the hidden state, whilst matrix W_{out} provides the mapping from the hidden-state vector space to the environment vector space. (c) Dynamics of hidden states of RNN, which have a form similar to the environmental dynamics but in a different vector space.*

The RNN architecture comprises the ability of storing information on the previous history in its hidden-state vector. The typical structure of RNN consists of input layer, hidden layer and output layer, within each of which layer a vector can represent the activation value of corresponding neuron nodes, namely $\vec{v}_{in}$, $\vec{s}$ and $\vec{v}_{out}$ for input layer, hidden layer and output layer respectively. A typical implementation of RNN can be stated as in the following equations:

$$\begin{aligned} \vec{s}_t &= \mathscr{H}(W_{hid}\vec{s}_{t-1} + W_{in}\vec{v}_{in}) \\ \vec{v}_{out} &= \mathscr{G}(W_{out}\vec{s}_t) \end{aligned}$$

where W_{hid} W_{in} W_{out} denote weight matrices of synaptic connections between input, hidden and output layers; activation functions are represented by symbols $\mathscr{H}$ and $\mathscr{G}$. For further information on recurrent neural networks, please refer to *chapter-2*. Note that the current value of hidden-state vector $\vec{s}_t$ is a function of its own measurement value at previous time step $\vec{s}_{t-1}$ and other information (in this case, the input data $\vec{v}_{in}$). It can therefore be regarded as an accumulation of its own history, and expressed using a mapping $\mathscr{H}$, in general: $\vec{s}_t = \mathscr{H}(\vec{s}_{t-1}, \cdots)$, as shown in *fig-1.1(c)*.

With the Recurrent Neural Network architecture as a black-box model, one can build the approximated dynamics model of the environment from the observed data. The main idea is to use the RNN model for converting the dynamics of the environment into the dynamics of the hidden states of the RNN model, and use the read-out matrix W_{out} as the interpretation method for the final output. Such a dynamics transformation is illustrated by *fig-1.1(b)*.

1.2.2 Evolution Framework

Evolutionary operations modify the structure of the learning model instead of merely modifying model parameter values. They cause therefore usually abrupt changes of the learning model in question, which as a result requires a learning model with fast convergence.

Benefits of using the evolution approach are (1) that it modifies the model structure in order to alter the dependence of the model on some data from outside, for instance the dependence of the neural network structure on one particular input data item; (2) that it reduces the complexity of the learning model because of constraints of the application and of the hardware configuration; (3) that it is robust enough to be able to adapt to drifts in the environment.

However, most of the available evolution methods are based on genotype selection within a large population. These evolution methods cannot be applied in sensor networks because of the constraints on processing speed and on storage memory as well as communication bandwidth *etc*. Hence, one is looking for an evolution method suitable for sensor network applications, with the aim that sensor nodes can exchange information, thus better adapting to the environment.

1.3 Structure of Thesis

This thesis is roughly divided into four parts: *chapter-2* describes the state-of-art on learning models and provides background on sensor network applications; the second part including *chapter-3* and *chapter-4* is concerned with a novel structure of RNN, including comparisons with conventional RNN models; the third part, including *chapter-5* and *chapter-6*, focuses on the *duty-cycle reduction* and *evolution framework* schemes developed for sensor network applications and related simulations; the last part is the conclusion in *chapter-7*.

chapter-1 (the current chapter) explains the target of this thesis and its challenges as well as the solution in general;

chapter-2 presents the current research in related topics, mainly in two categories including Recurrent Neural Network structure and evolution algorithms;

chapter-3 introduces a novel Recurrent Neural Network structure which is efficient and stable for long-term prediction tasks;

chapter-4 compares results from the novel neural network structure with those from conventional structures;

chapter-5 explains the reduction of energy consumption in sensor networks, based on the *duty-cycle reduction* scheme and on the *evolution framework* scheme;

chapter-6 presents evidence of improvement in sensor networks in terms of energy consumption, by using *duty-cycle reduction* and *evolution framework* schemes

chapter-7 concludes the thesis.

Chapter 2

State of the Art

Each sensor network application represents much knowledge and know-how from academia and industry, such as network topology, communication protocols, integrated circuits, and embedded learning models. In this chapter, some previous work concerned with sensor network applications will be discussed. The discussion will be limited to topics directly related to the solutions given in this thesis, as there are: neural network architectures and on-line learning algorithms, evolution algorithms and the background of sensor networks.

2.1 Recurrent Neural Networks

Modern neural network architectures, particular those used in off-line tasks, are complicated hybrid combinations of diverse neural structures [25, 26, 27, 28] with various statistics models [29, 30, 31, 32, 33]. In this section, due to the sensor nodes' limited processing capacity, neither such hybrid structures nor those evolutionary neural models [34, 35, 36, 37] will be discussed. The state-of-the-art neural network architectures mentioned in this section basically differ from each other in the structure of recurrent coupling topology inside the hidden layer. A gradient calculation for each neural architecture will also be considered, since it is one of the main factors of the learning model.

Before the discussion on recurrent neural networks (RNN), the concept of an artificial neuron will be given first. Computation with artificial neural networks was inspired by the functionality of a biological neural network, namely the human brain. Similar to a biological neuron, the artificial neuron has the structure as shown in *fig-2.1(a)* [38] where the neuron collects and accumulates the data $\vec{X} = [x_1, \ldots, x_5]^T$ from outside, then fires an output after applying a summation over a nonlinear function $g(x)$, called the *activation* function, with the summation sometimes being called the "netin" of neuron. This is formulated in *eq-(2.1)*, where b is the bias of the neuron, resembling the neuron's activation threshold.

$$y = g\left(\sum_i w_i x_i + b\right) \tag{2.1}$$

An artificial neural network is an ensemble of many artificial neurons, where in most cases groups of neurons activate synchronously and propagate their activation to another group of neurons. The classical example is the feed-forward operation between two layers of neurons as shown in *fig-2.1(b)*, with blacked-out circles representing neurons, a hollow circle denoting the bias "neuron" with activation value "1". Note that neurons within each group (layer) are usually independent from each other. *Eq-(2.2)* depicts such a situation of activation of the neuron-layer. A classical artificial neural network basically consists of groups of neurons connected from one group to another, in such a way that information is propagated.

$$y_i = g\left(\sum_j w_{i,j} x_j + b_i\right), \quad \forall i \tag{2.2}$$

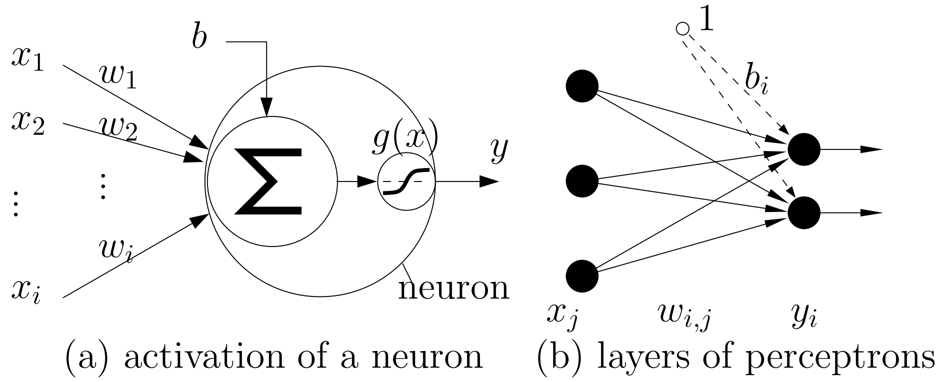


Figure 2.1: *The activation of an artificial neuron and the feed-forward between layers of neurons. (a) Activation of a neuron with input data from outside, where the summation of input and bias will be subjected to the nonlinear function $g(x)$. (b) Synchronization of activation between layers of neurons, with black circles as neurons and a hollow circle as the bias "neuron" with activation value "1".*

A recurrent neural networks (RNN) is a special class of artificial neural networks whose neurons within one layer can be directly or indirectly independent from each other, so that historical information can be stored in these self-coupling neurons. Forwarding in recurrent neural networks in general can be expressed as in *eq-(2.3)* and *eq-(2.4)*:

$$\mathbf{s}_t = \mathscr{H}(W_t, \mathbf{s}_{t-1}, \mathbf{x}_{t-1}) \tag{2.3}$$
$$\mathbf{x}_t = \mathscr{G}(W_t, \mathbf{s}_t) \tag{2.4}$$

where $\mathscr{H}$ is the mapping from input data $\mathbf{x}_{t-1}$ and the previous hidden-state vector $\mathbf{s}_{t-1}$ to the current hidden state vector $\mathbf{s}_t$, and $\mathscr{G}$ is the mapping from $\mathbf{s}_t$ to the network's final output $\mathbf{x}_t$. Such iterative forwarding will partially accumulate the information from vector $\vec{x}$ and store it in the hidden state vector $\vec{s}$, such that $s_t \dashleftarrow \{x_{t-1}, x_{t-2}, \cdots\}$ where symbol $\dashleftarrow$ represents the data-fusion direction. From this information pool, represented by $\vec{s}$, the output mapping can render a precise report to the network output. Such rendering mapping $\mathscr{G}$ is normally a simple linear function, with or without a squashed function on top of it. Therefore, most RNN models differ from each other in the mapping $\mathscr{H}$, consequently in the topology of hidden-neuron connections or the structure of the hidden layer. A sketch of RNN in general is given in *fig-2.2*. In the following text, several RNN models will be discussed focusing on the difference in structure of the hidden layer.

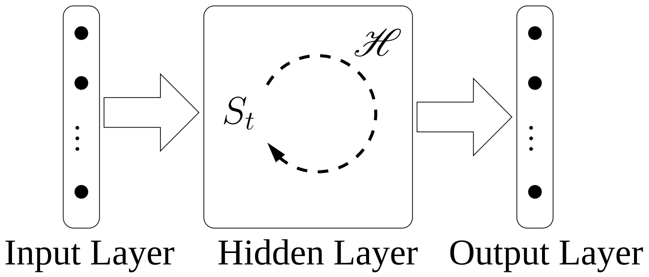


Figure 2.2: *Sketch of the mechanism of recurrent neural network models in general, whose main contribution lies in the recurrent coupling in the hidden layer, where historical information is stored.*

In the following, W_{loc} is used to denote the synaptic links in different locations, *e.g.* W_{in} denotes the values of connection weights from the input layer to the hidden layer, W_{hid} denotes the values of recurrent connection inside the hidden layer and W_{out} denotes those value of connections from the hidden layer to the output layer. Symbol b denotes the bias of hidden and output neurons. All of W_{loc}, b and the fixed weights[1] together constitute the vector of system parameters W_t in *eq-(2.3)* and *eq-(2.4)*. Note that only those trainable weights will be taken into account in the calculation of the gradient matrix (*i.e.* network output gradient *w.r.t.* trainable weights), and hence influence the computational cost of the learning algorithms.

In this section, diagrams have been employed to depict the structure of different neural network models. If nothing else if stated, black dots represent the neuron nodes of neural network, arrows are synaptic connections between neurons or between layers (dashed arrows are connections under training; solid arrows are constant connections with fixed weights). Group of similar neurons are enclosed in a block circled with closed dash-dot line, in order to show the scope of mappings $\mathscr{H}$ and $\mathscr{G}$. Depending on the size a block, connections from or to a block of neurons will be full connections of a m-to-n mapping but, for the sake

[1] In this thesis, fixed weights occur only in the echo state network model which will be discussed later.

of simplicity, only a few of such connections are drawn. Connections from one neuron to another will always be 1-to-1 connections.

Time Delay Neural Networks (*TDNN*):

The classic multi-layer perceptron (*MLP*) architecture doesn't have recurrent connections, but it is worth discussing since all RNN models are based on or inspired by this structure. The *MLP* model has been mainly used for static regression or classification problems [39, 40, 41]. However, extensions of *MLP* have also been applied to temporal tasks such as time series prediction and time series classification [42, 43, 22]. Besides current available data, a typical extension uses also previous data as network input in order to turn the temporal sequence into spatial patterns on the input layer of the network. Such a structure is called a time-delay neural networks (*TDNN*) model [22, 23, 44]. For sufficient modeling power, the *TDNN* should have at least 3 layers. Otherwise it would be basically a linear autoregressive model with additional squashing of the outputs by a saturating transfer function such as the "sigmoid" function. With $j = [1, \ldots, d]$ labeling the different components of the d-dimensional data and τ denoting the number of previous data to be memorized for modeling, the 3-layer *TDNN* architecture has the following update equations:

$$\begin{aligned}
\mathbf{s}_t &= \mathscr{H}(\mathbf{s}_{t-1}, \mathbf{w}_t, \mathbf{x}_{t-1}) \\
\mathbf{x}_t &= \mathscr{G}(\mathbf{s}_t, \mathbf{w}_t) = g(a_g) \\
&= g\left(W_{out} h(a_h) + \mathbf{b}_2\right) \\
&= g\left(W_{out} h(W_{in}\mathbf{s}_t + \mathbf{b}_1) + \mathbf{b}_2\right)
\end{aligned}$$

where $\mathscr{H}(\ldots)$ is realized as follows:

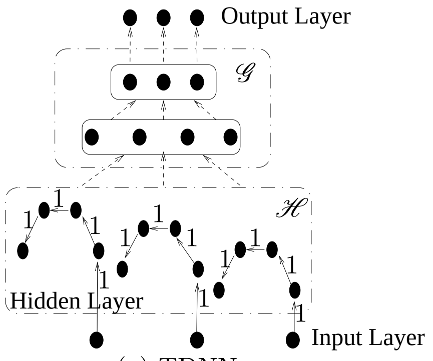
$$\mathbf{s}_{t,i+(j-1)*\tau} = \begin{cases} \mathbf{s}_{t-1,i+1+(j-1)*\tau} & \text{if } i = 1, \ldots, \tau - 1 \\ \hat{\mathbf{x}}_{t-1,j} & \text{if } i = \tau \end{cases}$$

In this model, function $h(\ldots)$ is the identity map; activation function $g(\ldots)$ is the squash function "tanh", where it is understood that for vector arguments the squashing function is applied to each vector component individually. The mapping $\mathscr{H}(\ldots)$ in the *TDNN* model is just a shift register for input vectors and is not subject to learning. *Fig-2.3(a)* shows the typical structure of such a *TDNN* architecture for a 3-dimensional time series with $\tau = 4$, where only the two-layered output mapping $\mathscr{G}$ has to be trained.

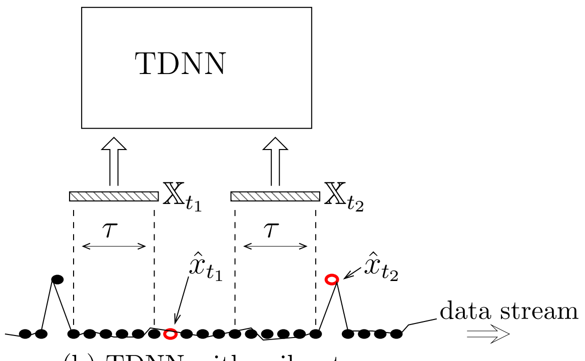
The calculation of gradient Ψ in *TDNN* is given by the following equations:

$$\Psi = \frac{d\mathbf{x}_t}{d\mathbf{w}} = \frac{dg}{da_g}\left(\frac{\partial a_g}{\partial \mathbf{w}} + \frac{\partial a_g}{\partial \mathbf{h}}\frac{dh}{da_h}\frac{\partial a_h}{\partial \mathbf{w}}\right)$$

The main drawback of the *TDNN* structure is the need to fix the time window for temporal memory in advance, *i.e.* the value of the parameter τ. This requires some prior knowledge about the environment which contradicts the requirement $< \Re 4 >$ on *page-3*. In the case of an insufficient τ value, the *TDNN* model even fails to modulate the dynamics of a simple spike time series (ref. *section-4.1.1*). As shown in *fig-2.3(b)*, black dots at the bottom represent the data stream values at successive time steps. The two shaded bars represent the input data $\mathbb{X}_{t1}$ and $\mathbb{X}_{t2}$ of time steps $t1$ and $t2$ respectively, and two hollowed red dots represent the corresponding target values $\hat{x}_{t1}$ and $\hat{x}_{t2}$ of *TDNN*. Because of the character of the spike time series, there is no difference in input data $\mathbb{X}_{t1}$ and $\mathbb{X}_{t2}$, as shown in the figure. Being a static modeling architecture, the *TDNN* model produces outputs of the same value and fails to predict the approaching spike $\hat{x}_{t_2}$.



(a) TDNN



(b) TDNN with spike stream

Figure 2.3: *(a) Example of a* TDNN *model, with 3-dimensional data and the number of data-patterns going up to 4. Note that the values of connection weights in the pseudo hidden layer are constant and equal to 1, which means that history is simply duplicated and shifted. (b) Example of a* TDNN *model being trained with a spike time stream. Black dots at the bottom denote data stream values with two peaks in the time window; the shaded bar represents data of length τ collected from the input data stream to the* TDNN *model, where $\mathbb{X}_{t1}$ and $\mathbb{X}_{t2}$ are inputs at different time steps, their corresponding targets being $\hat{x}_{t1}$ and $\hat{x}_{t2}$.*

Simple Recurrent Nets (*SRN*):

Unlike a shift register in *TDNN*, a recurrent neural network (RNN) model has recurrent couplings. Thus it has the theoretical ability to embed an infinite history within its recurrent hidden layer, constituting in effect an infinite impulse filter [45]. Elman [46] introduced simple recurrent neural networks (*SRN*) with a duplicated context unit of the hidden layer in *MLP*, which at the core constituted a fully connected hidden layer of a RNN as shown in *fig-2.4(a)*. Since then, RNNs have been applied successfully in many applications, but mostly they were trained off-line [28, 47].

The discrete update equations of a *SRN* are:

$$\begin{aligned}\mathbf{s}_t &= \mathscr{H}(\mathbf{s}_{t-1}, \mathbf{w}_t, \mathbf{x}_{t-1}) = h(a_h) \\ &= h(W_{hid}\mathbf{s}_{t-1} + W_{in}\hat{\mathbf{x}}_{t-1} + \mathbf{b}_2) \qquad (2.5)\\ \mathbf{x}_t &= \mathscr{G}(\mathbf{s}_t, \mathbf{w}_t) = g(a_g) \\ &= g(W_{out}\mathbf{s}_t + \mathbf{b}_1) \qquad (2.6)\end{aligned}$$

where $g(\dots)$ is the identity output activation function and $h(\dots)$ is the "tanh" hidden activation function; W_{hid}, W_{in} and W_{out} are the corresponding synaptic weights respectively; $\mathbf{b}_1$ and $\mathbf{b}_2$ are bias parameters of output and hidden neurons.

According to *eq-(2.6)* and *eq-(2.5)*, the gradient Ψ is calculated as follows:

$$\begin{aligned}\frac{d\mathbf{s}_t}{d\mathbf{w}} &= \frac{dh}{da_h}\left(\frac{\partial a_h}{\partial \mathbf{w}} + W_{hid}\frac{d\mathbf{s}_{t-1}}{d\mathbf{w}}\right)\\ \Psi = \frac{d\mathbf{x}_t}{d\mathbf{w}} &= \frac{dg}{da_g}\left(\frac{\partial a_g}{\partial \mathbf{w}} + W_{out}\frac{d\mathbf{s}_t}{d\mathbf{w}}\right)\end{aligned}$$

Note that calculation of $\frac{d\mathbf{s}_t}{d\mathbf{w}}$ is based on the RTRL method, therefore the previous derivative of state vector *w.r.t.* model parameters is stored at each time step.

The fully occupied matrix W_{hid} of recurrent synaptic weights will be trained. As matrix W_{hid} is basically unbounded, on-line training may modify the matrix value such that the recurrent layer could become dynamically unstable. Thus it could happen that learning doesn't converge or is trapped in a bad solution, contradicting to the requirement $< \Re 4 >$ on *page-3*.

Echo State Neural Networks (*ESN*):

Jaeger [48, 49] proposed the echo state neural networks (*ESN*), aiming at simple learning while trying to avoid dynamic instabilities. *ESN*s can have the same topology as *SRN*s, as shown in *fig-2.4*. However, in this approach, values of entries of W_{hid} and W_{in} are kept fixed and initialized with random values; furthermore the former is re-scaled such that its largest absolute eigenvalue does not exceed a predefined limit $\lambda_{max} < 1$; this "echo state" property of the hidden layer guarantees the diminishing influence of past states and consequently avoids instabilities in the dynamics of the recurrent hidden layer. This property of *ESN*s is responsible for its name because input information is propagated through the hidden layer like an echo, becoming weaker with each iteration. Despite this "echo-state" property of the recurrent hidden layer, *ESN* models are able to simulate *e.g.* chaotic time series. This can be achieved by an outer feedback looping from the output of the network to its input.

As the only parameters which are subject to learning in the *ESN* model are the weights W_{out} of the linear output mapping, this awards the *ESN* model a simple learning process. The update equation for the hidden states of *ESN*s is similar to *eq-(2.5)* for *SRN*s, except that the matrices W_{in} and W_{hid} are merged into one single

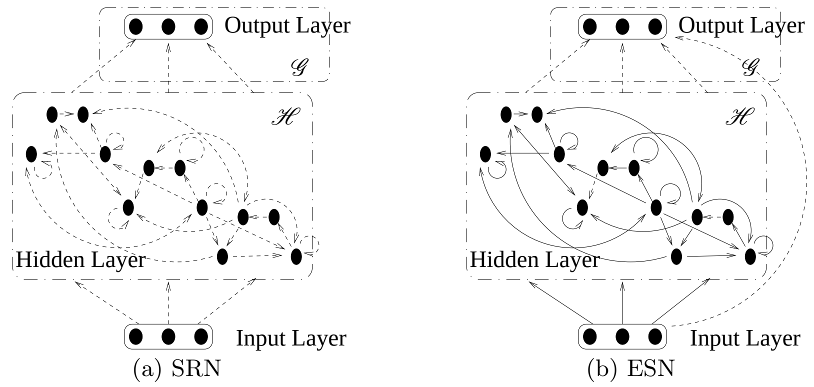


Figure 2.4: *The* SRN *and* ESN *models, whose topology can be identical except for the difference in the variable-or-constant condition of connection weights. Both of them possess a hidden layer with random topology. (a) The* SRN *model with variable connection weights, particularly that of mapping $\mathscr{H}$, can adjust the mapping $\mathscr{H}$ accordingly, though such unconstrained mappings can make system unstable; (b) The* ESN *model with constant connection weights except in output mapping $\mathscr{G}$. Manual pre-tuning of the eigenvalue-spectrum of mapping $\mathscr{H}$ makes the* ESN *model stable and efficient, though leaving mapping $\mathscr{H}$ unchanged during training can reduce the robustness of the model.*

matrix W_{fix} indicating that the values of its entries are fixed. The update of the output states obeys *eq-(2.11)*.

$$\begin{aligned}
\mathbf{s}_t &= \mathscr{H}\left(\mathbf{s}_{t-1}, \mathbf{w}_t, \mathbf{x}_{t-1}\right) = h(a_h) && (2.7)\\
&= h\left(W_{hid}\mathbf{s}_{t-1} + W_{in}\hat{\mathbf{x}}_{t-1} + \mathbf{b}_2\right) && (2.8)\\
&= h\left(W_{fix}\begin{bmatrix}\mathbf{s}_{t-1}\\ \hat{\mathbf{x}}_{t-1}\end{bmatrix} + \mathbf{b}_{fix}\right) && (2.9)\\
\mathbf{x}_t &= \mathscr{G}(\mathbf{s}_t, \mathbf{w}_t, \mathbf{x}_{t-1}) = g(a_g) && (2.10)\\
&= g\left(W_{out}\begin{bmatrix}\mathbf{s}_t\\ \hat{\mathbf{x}}_{t-1}\end{bmatrix}\right) && (2.11)
\end{aligned}$$

where W_{fix} and $\mathbf{b}_{fix}$ are now the prefixed synaptic hidden weights and bias, and they will not be considered in training. *Fig-2.4(b)* shows the structure of a *ESN*, where solid lines represent synaptic connections with constant weights whilst dashed lines represent connections to be trained. The gradient Ψ for the learning algorithm has a very simple form:

$$\Psi = \frac{d\mathbf{x}_t}{d\mathbf{w}} = \frac{dg}{da_h}\frac{\partial a_h}{\partial \mathbf{w}} \qquad (2.12)$$

The *ESN* approach requires a recurrent layer of large size in order to obtain a rich reservoir of dynamic states in the hidden layer. The size of the hidden layer depends mainly on the complexity of the learning task. Similarly, the limit value λ_{max} has to be carefully chosen in advance to obtain better results; however, the proper choice of λ_{max} is less critical than the one of the size of its recurrent hidden layer. It should not be too small to store enough historical information in the hidden states. A typical value for λ_{max} is 0.8 [50].

The fact that values of input and recurrent connection weights in *ESN*s are constant has simplified the training procedure of the model and reduced the computational cost per iteration. However the fact that only the output connections are trainable can, on the other hand, degrade the generalization ability of the model, which will be proven in *Chapter-4*. In fact, as suggested by Jaeger [51], manually tuning the λ_{max} value in each application is important for the *ESN* model in order to achieve the best performance. This requires *a priori* knowledge of the application and data, in contradiction to requirements $< \mathfrak{R}1 >$ and $< \mathfrak{R}2 >$ given on *page-3*.

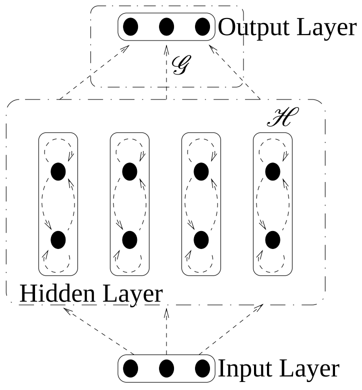


Figure 2.5: *The structure of* BDRNN*s. Inside the hidden layer, every pair of neurons constitutes a block matrix in hidden weights and they are isolated from the other hidden neurons. Note that each sub-block has the same structure.*

Block-Diagonal RNNs (*BDRNN*):

Sivakumar *et al.* [52] introduced the *BDRNN* model which possesses a hidden-weight matrix of block-diagonal structure. Each sub-block matrix is of size 2×2. Among variants of this block-diagonal structure, the scaled orthogonal version of the *BDRNN* is superior to the free-form version *BDRNN* [52, 53], where the former one consists of scaled orthogonal sub-matrices, whilst the latter one doesn't have any constraint (*i.e.* entries of the matrix can theoretically have arbitrary value). In the scale orthogonal version, sub-matrix $w \in \mathbb{R}_{2\times 2}$ and its entries $w = \{w_{1,1}, w_{1,2}; w_{2,1}, w_{2,2}\}$ satisfy the following formulas:

$$w_{1,1} = w_{2,2}; w_{1,2} = -w_{2,1};$$
$$w_{1,1}^2 + w_{1,2}^2 \leq 1.0$$

Therefore, the scaled orthogonal *BDRNN* model has feature similar to the *ESN* model, namely that the maximum absolute eigenvalue of the hidden-weight matrix is limited, such that $\lambda_{max} \leq 1$. It is reported in [52] that the scaled orthogonal *BDRNN* with constrained eigenvalue spectrum has outperformed the free version (without constraint) *BDRNN*.

The update equation of the *BDRNN* is similar to that of the *SRN*:

$$\begin{aligned}\mathbf{s}_t &= h(W_{hid}\mathbf{s}_{t-1} + W_{in}\hat{\mathbf{x}}_{t-1} + \mathbf{b}_2)\\ \mathbf{x}_t &= g(W_{out}\mathbf{s}_t + \mathbf{b}_1)\end{aligned}$$

though with a different structure of W_{hid}, as shown in *fig-2.5*. Due to the paired structure, the number of hidden neurons in *BDRNN*s has to be even. The scaled orthogonal DBRNN model resembles *ESN*s in that the maximum absolute eigenvalue of W_{hid} is limited to $\lambda_{max} \leq 1$.

2.2 On-line Learning Algorithms

Considering the presentation of training data, training methods for neural network models can be roughly categorized in two classes: batch training methods and on-line training methods. In batch training, the neural network model, in each iteration, is trained with a batch of data items at one time, such that the change $\triangle_{\mathbf{w}}$ of model parameter $\mathbf{w}$ is derived from the data set $\{\hat{x}_1, \ldots, \hat{x}_n\}$:

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \triangle_{\mathbf{w}} \mid \{\mathbf{w}_{t-1}, \hat{x}_1, \ldots, \hat{x}_n\} \tag{2.13}$$

By contrast, on-line learning adjusts the model parameter in each iteration with the increment $\triangle_{\mathbf{w}}$ which depends on the current individual data, though in some cases also depending on the memory vector s_t, as in *eq-(2.14)*. Memory vector s_t holds the history of input data, and can be realized as a hidden state vector in the recurrent neural network case.

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \triangle_{\mathbf{w}} \mid \{\mathbf{w}_{t-1}, s_t, \hat{x}_t\} \tag{2.14}$$

Batch training methods using a finite training set in each iteration have generally demonstrated their ability to provide acceptable results. However, they requires significant computation power and massive amounts of memory, both unavailable or unaffordable in sensor network applications. On the other hand, on-line training methods have simplified the training process by taking only one data set into account in every iteration step. There is much empirical evidence that on-line learning can restore the trajectory of the batch learning model and find the minimum attractor imposed by the training data [54].

Many of the neural network learning algorithms, presented in this section, can easily be converted to its batch-mode for off-line use (some were originally introduced as batch-mode algorithms and were later modified to on-line ones). But as specified by the requirements in *section-1.2.1* on *page-3*, only their on-line versions are addressed here.

In spite of the difference in learning algorithms, error functions of neural network applications will normally take the form as in *eq-(2.15)*, where the evaluation error E measuring the Euclidean-like distance between output and target depends on parameters $\mathbf{w}$, symbol $\hat{y}$ stands for the corresponding target (the network output y can be expressed as a function of $\mathbf{w}$ such that $y = f(\mathbf{w})$), and the coefficient $\frac{1}{2}$ is used for convenience of calculation.

$$E[\mathbf{w}] \;=\; \frac{1}{2}\sum_{i=1}^{d}\left(\hat{y}_i - y_i\right)^2 = \frac{1}{2}\sum_{i=1}^{d}\Big(\hat{y}_i - [f(\mathbf{w})]_i\Big)^2 \tag{2.15}$$

With the definition of residual δ:

$$\delta_i = \hat{y}_i - y_i = \hat{y}_i - [f(\mathbf{w})]_i,\ \forall i, \tag{2.16}$$

the associated gradient calculation is given by *eq-(2.17)*:

$$\frac{\partial E[\mathbf{w}]}{\partial \mathbf{w}} \;=\; -\sum_{i=1}^{d}\delta_i\frac{\partial y_i}{\partial \mathbf{w}} = -\sum_{i=1}^{d}\delta_i\frac{\partial [f(\mathbf{w})]_i}{\partial \mathbf{w}} \tag{2.17}$$

2.2.1 Gradient Descent Learning

Being able to start from any starting point, gradient descent learning adjusts its parameters, as the name says, by following the error gradient. Given a task with an evaluation error E as defined in *eq-(2.15)* and also a gradient as in *eq-(2.17)*, the parameter adjustment step is given in *eq-(2.18)* and the parameters are updated using *eq-(2.19)*.

$$\Delta w_j \;=\; -\eta\frac{\partial E}{\partial w_j} = \eta\sum_{i=1}^{d}\delta_i\frac{\partial y_i}{\partial w_j} \tag{2.18}$$

$$w_j \;=\; w_j + \Delta w_j \tag{2.19}$$

where $\eta \in \mathbf{R}^+$ is the predefined learning rate (usually chosen very small), j is the index, and δ_i is the corresponding residual for index i.

2.2.2 Back-Propagation

The back-propagation learning algorithm for multi-layer perceptron (*MLP*) is a further development of gradient descent learning; it is very successful in research and application.

Basically, back-propagation learning can be divided into two steps: the feed-forward step and the backward propagation step. Feed-forward of the *MLP* model involves conveying the information $\vec{x}$ from the input layer through the hidden layer ($\vec{s}$) to the output layer, where output value $\vec{y}$ is supposed to coincide with target value $\hat{y}$, though in most cases they are not equal. In the backward propagation step, the residual of the network output, *i.e.*

$$\delta^\dagger = \hat{y} - \vec{y}, \tag{2.20}$$

is fed backwards through the same synaptic connections, modifying the associated connection weights accordingly. These two steps are depicted in *fig-2.6* where solid lines represent the feed-forward step and dashed lines represent back-propagation step.

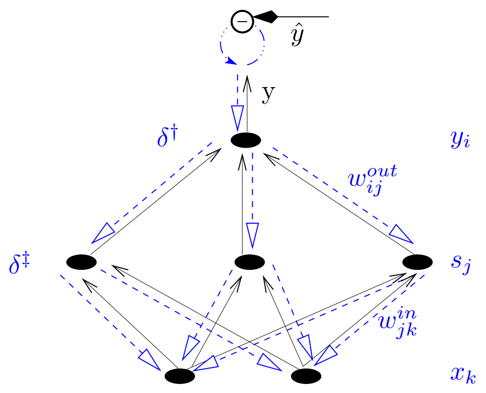


Figure 2.6: *Back-propagation of* MLP*. The* MLP *model in the example has three layers, namely input, hidden and output layer. Inside these layers, they have {2,3,1} neuron nodes (black dots) respectively. Arrows show the direction of information flow: solid lines represent the feed-forward step and dashed lines represent the backward propagation step. Output and target are respectively denoted by* y *and* $\hat{y}$*,* x *and* s *represent the input and hidden state of the network,* $\delta^\dagger$ *is the corresponding residual for the output layer,* $\delta^\ddagger$ *is the virtual residual for the hidden layer. The entries in matrices of the network connections between hidden layer and output layer are given as* w_{ij}^{out} *and those between input and hidden layer are given as* w_{jk}^{in}*, where* i, j, k *are indices.*

For the sake of simplicity, it is assumed that a three-layer *MLP* model has linear activation functions for all neurons, and it uses the update equations in *eq-(2.21)* and *eq-(2.22)*, where i, j, k are the index for output, hidden and input layer respectively.

$$s_j = \sum_k w_{jk}^{in} x_k \tag{2.21}$$

$$y_i = \sum_j w_{ij}^{out} s_j \tag{2.22}$$

The calculation of modification for weight w_{ij}^{out} from hidden layer to output layer is given below in *eq-(2.23)*. For weight w_{jk}^{in} from input layer to hidden layer the calculation can be expressed in a similar way as in *eq-(2.24)*. Generally, the back-propagation algorithm first guesses the direct-output residual of weights belonging to one particular layer and then adjusts the respective parameters in the same layer accordingly.

$$\Delta w_{ij}^{out} = -\eta \frac{\partial E_i}{\partial w_{ij}}^{out} = \eta \delta_i^{\dagger} s_j \tag{2.23}$$

$$\begin{aligned}\Delta w_{jk}^{in} &= -\eta \frac{\partial E}{\partial w_{jk}^{in}} = -\eta \sum_{i=1}^{d} \frac{\partial E_i}{\partial s_j} \frac{\partial s_j}{\partial w_{jk}^{in}} \\ &= \eta \sum_{i=1}^{d} \delta_i^{\dagger} w_{ij}^{out} x_k = \eta \delta_j^{\ddagger} x_k\end{aligned} \tag{2.24}$$

$$\text{where} \quad \delta_j^{\ddagger} = \sum_{i=1}^{d} \delta_i^{\dagger} w_{ij}^{out} \tag{2.25}$$

2.2.3 Kalman Filters

A Kalman filter [55, 56, 57] is an optimal estimator, which estimates the value of state x_t of a discrete linear system in *eq-(2.26)* and *eq-(2.27)*, developing dynamically over time, where x_t, the state of dynamics, is emulated in *eq-(2.26)* and z_t, the observation of state, is given by *eq-(2.27)*. A, B, H are constant matrices referring to the static parameters of the system. Symbols μ_t and ν_t represent the processing noise and measurement noise which respectively satisfy the zero-mean normal distributions: $\mu_t \sim \mathscr{N}(0, Q_t)$ and $\nu_t \sim \mathscr{N}(0, R_t)$.

$$x_t = Ax_{t-1} + Bu_t + \mu_t \tag{2.26}$$

$$z_t = Hx_t + \nu_t \tag{2.27}$$

Solving the mentioned dynamics system with a Kalman filter requires two steps, namely the *time update* updating the state according to the given model, and the *measurement update*, correcting the model obtained from the observation.

In the *time update* step, state value x_t and its covariance matrix P_t get updated according to the stochastic system in *eq-(2.26)*:

$$P_t^{\dagger} = AP_{t-1}A^T + Q_t \tag{2.28}$$

$$x_t^{-} = Ax_{t-1} + Bu_t \tag{2.29}$$

where x_t^- and $P_t^{\dagger}$ are respectively the temporal values for the estimated state and its covariance, Q_t is the covariance matrix of noise μ_t. When the observation z_{t+1} is available, its state value and covariance will be corrected according to:

$$P_t = \left(\left(P_t^{\dagger} \right)^{-1} + H^T R_t^{-1} H \right)^{-1} \tag{2.30}$$

$$x_t = x_t^- + P_t H^T R_t^{-1} \left(z_t - Hx_t^- \right) \tag{2.31}$$

Extensions of this method have been successfully applied to non-linear system and on-line learning [58, 59, 60, 61, 62, 63, 64, 65]. Employing the extended Kalman filter (EKF) in training of recurrent neural networks, x_t refers to the network parameters which are to be optimized, H refers to the gradient of observation (*i.e.* the target data corresponding to the network output) *w.r.t.* parameters. Looking for a stable model in the context of this thesis, parameters of a well trained network should stay close to the target parameter-set. Therefore, matrix A in *eq-(2.26)* is the identity matrix and the value of u_t is set to $u_t = 0$ representing the autonomy of dynamics without outer control force.

2.2.4 Learning Paradigms

Real-time recurrent learning

Real-time recurrent learning (RTRL), introduced by Williams and Zipser [66], has been applied in many projects concerned with recurrent neural nets[67, 68, 69] where an on-line version is available without requirement on large amount of memory. The RTRL method is specific for recurrent neural networks where recurrent coupling exists in the hidden layer. In RTRL it is assumed that the values of the hidden-state vector depend also on the weights, and should also contribute to the gradient calculation. Therefore, unlike back-propagation learning where gradient *w.r.t.* the weights is inversely propagated from output layer to input layer, RTRL requires the hidden-state gradient *w.r.t.* network parameters. Recalling the general evolving equation of hidden-state in recurrent neural networks:

$$\mathbf{s}_t = \mathscr{H}(W_t, \mathbf{s}_{t-1}, \mathbf{x}_{t-1}) \tag{2.32}$$

the calculation obeys *eq-(2.33)*, where $\mathscr{U} = \{W_t, x_t\}$ denotes the other variables besides the hidden-state vector $\mathbf{s}$.

$$\frac{\partial \mathbf{s}_t}{\partial W} = \frac{\partial \mathscr{H}}{\partial \mathbf{s}_{t-1}} \frac{\partial \mathbf{s}_{t-1}}{\partial W} + \frac{\partial \mathscr{H}}{\partial \mathscr{U}} \frac{\partial \mathscr{U}}{\partial W} \tag{2.33}$$

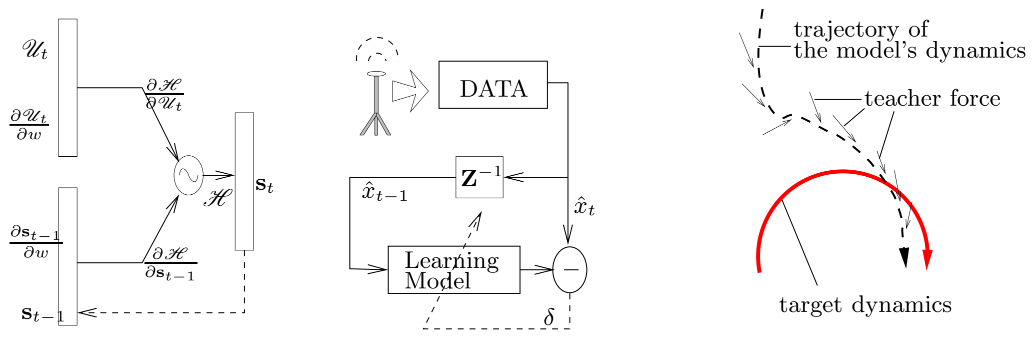
With the initial condition:

$$\frac{\partial \mathbf{s}_0}{\partial W} = 0 \tag{2.34}$$

reflecting that the initial value of hidden-state vector is independent from the network weight, the RTRL method manages the retention of gradient information over a time period of either fixed or indefinite length by iterating the gradient calculation in *eq-(2.33)* at each time step. A sketch of RTRL paradigm is given in *fig-2.7(a)*, where the dash line represents the feedback from the context unit.

Teacher Forcing

Teacher Forcing [71, 72, 70] refers to the learning paradigm where learning is driven



$\mathscr{U}_t$
$\frac{\partial \mathscr{U}_t}{\partial w}$
$\frac{\partial \mathscr{H}}{\partial \mathscr{U}_t}$
$\mathbf{s}_t$
$\mathscr{H}$
$\frac{\partial \mathbf{s}_{t-1}}{\partial w}$
$\frac{\partial \mathscr{H}}{\partial \mathbf{s}_{t-1}}$
$\mathbf{s}_{t-1}$
DATA
$\mathbf{Z}^{-1}$
$\hat{x}_{t-1}$
$\hat{x}_t$
Learning Model
$-$
δ
trajectory of the model's dynamics
teacher force
target dynamics

(a) RTRL (b) Teacher Forcing (c) Force of Dynamics

Figure 2.7: *(a) Real time recurrent learning paradigm, where the hidden-state gradient w.r.t. weights at the previous time step will contribute to the calculation at the current time step; (b) Teacher forcing leaning paradigm (adapted from [70]) (c) Pursuit of dynamics by teacher forcing, with a solid red line indicating the target dynamics and a dashed line indicating the actual dynamics of the model which is always moving towards the target by means of the push from the data (short solid arrow line).*

by data input from outside, such that the input value is more informative in terms of environmental dynamics. This has been shown to be an effective technique for a training algorithm. A schematic sketch of teacher forcing is shown in *fig-2.7(b)*. The dashed curve represents the trajectory of dynamics of the learning model, gradually being adjusted by the teacher force and approaching the target dynamics imposed by the data.

2.3 Backgrounds on Sensor Network Application

Smart sensor nodes (with "intelligence") contain not only the measurement unit but also a micro-controller unit (MCU) [2] for data processing, a memory unit for data storage and a radio unit for communication. Operations as well as activations of these units will take a certain amount of time and energy, which has to be considered when designing the sensor network. For example, the TelosB[73] is an ultra-economical wireless sensor module launched in 2004 by the University of California at Berkeley. TelosB sensor node is capable of transmitting data at up to 250 kbps and of handling sophisticated data-processing applications. *Table-2.3* gives some specifics on the TelosB's energy handling. Note that radio communication draws far more current than the other operations.

[2] A micro-controller (MCU) in a sensor node resembles the CPU in a personal computer, but it is highly optimized regarding overall operating cost. Its design aims to keep the number of external components and connections low, which is why it has substantial on-chip ROM and RAM.

Operation	Draw	Description
Device Standby	5.1 μA	Sleeping
MCU Idle	54.5 μA	After wakeup but idle
MCU Active	1.8 mA	Data processing
MCU + Radio RX	21.8 mA	Communication
MCU + Radio TX	19.5 mA	Communication
MCU Wakeup time	6 μs	Wakeup
Radio Wakeup time	580 μs	Wakeup

Table 2.1: *Specifications of TelosB sensor. "Radio RX" refers to receive data, "Radio TX" refers to data transmission. Symbol "m" means* 10^{-3} *and "µ" means* 10^{-6}*. "A" (Amperes) is the unit for current and "s" (seconds) is the unit for time.*

As sensor nodes in distributed sensor networks are usually supported by un-replaceable batteries, sensor nodes will usually be turned into sleep mode, in order to preserve energy, after their current work has been finished and will be re-activated when a wake-up event happens. In sensor network applications, the full cycle from activation to sleep and to re-activation is called a *working cycle*.

Details of the working cycle depend on the device's hardware and the requirements of the application. *Fig-2.8* depicts a typical working cycle of a TelosB sensor node. At the beginning of the working cycle, the sensor is woken up as scheduled or triggered by a wake-up event. The time for wake-up of the MCU in TelosB is about $6\mu s$, and wake-up of radio in TelosB takes $580\mu s$. Having the sampled data, data transmission is carried out next, where the transmission serves data analysis purposes . After collecting all required data from neighbors, the smart sensor will trigger the learning process and improve the embedded model slightly. This can conclude the active time of the sensor node. In some cases, the sensor node may be asked to perform a prediction on its measurements where the prediction serves for further diagonal purpose. After this, the sensor node will go into sleep model till a wake-up condition is satisfied again.

As energy management is important in sensor networks, one of the energy consumption criteria is the so-called "duty cycle", *i.e.* the fraction of time during which the sensor is active and working, defined as $\frac{\tau}{T}$ (refer to *fig-2.8*).

2.4 Evolution

Computer algorithms derived from evolution theory have been caught much attention, particular from the computer games industry. The evolutionary algorithm (EA) is a generic

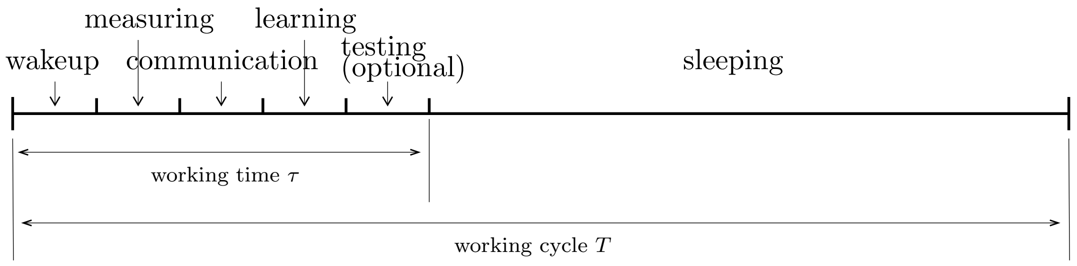


Figure 2.8: *Typical working cycle of a sensor, consisting of "wake-up", "sampling", "evolution & transmit", "learning", "testing" and "sleeping". Note that "testing" is only for evaluation purposes. Symbol τ represents the total time in each cycle where the device is activated, T represents the entire working cycle time. In general, the duty cycle is calculated as the ratio of τ divided by T.*

population-based heuristic optimization algorithm. Inspired by Darwin's biological evolution theory, the EA method involves similar operations, such as reproduction, mutation, crossover and selection.

The state-of-the-art methods of EA, including those presented further below, are normally based on a large population of samples, which is hard to mirror in a sensor network application because of the limitations in memory and processing capacity. However, evolution concepts, *e.g.* selection and crossover, behind these EA methods have inspired the development of the *evolution framework* presented in *chapter-5*.

2.4.1 Genetic Algorithm

The genetic algorithm (GA) is a search algorithm aiming to find the global optimum parameter set. In GA, parameters of the function to be optimized are encoded in a genotype vector and the evaluation of parameters is based on the fitness value of a particular set of parameters. Optimization means in this case: finding the representation of the genotype vector providing the best fitness value.

Inspired by evolutionary biology, the genetic algorithm also contains elementary operations of related concepts, including: inheritance, mutation, selection and crossover. In each generation, a population of sample individuals (*i.e.* candidate set of parameters) is given, and the fitness value of each individual in the population will be calculated. The selection procedure consists in choosing, based on the fitness value, the best or a group of the best individuals. They are the competition winners, and survive while all the other individuals are discarded. A new population, the next generation, can be formed by implementing mutation or crossover over the winners. The new generation will take part in the selection procedure of the next GA iteration.

2.4.2 Evolution Strategies

Evolution Strategies (ESs) [74, 75] are a family of search methods that based on the population of candidate solutions. Its evolutionary operations are similar to the genetic algorithm and include selection, crossover and mutation.

Each individual (parents included) in the population has a tuple of properties: $\{y, s, F(y)\}$, where vector y is an object parameter (*i.e.* a parameter which has to be optimized), s is a set of strategy parameters, $F(y)$ is the fitness value of parameter set y.

In each generation, a group of parents of size μ is initialized or inherits from the previous generation: $P = p_1, p_2, \ldots, p_\mu$. From the entire parent population, ρ parents $p_i \in P,\ i = [1, \ldots, \rho]$ are chosen to produce each offspring. Production of offspring should begin with the mutation of strategy setting s and then parameter-set y is formed based on the strategy setting s. Once the offspring generation is complete, its fitness value is evaluated, based on which the parents of next generation are selected.

Based on the number of parents, individuals in each population, ESs is usually expressed in a general form: $(\mu + \lambda) - ES$ or $(\mu, \lambda) - ES$. The difference between these two forms lies in the members of the population from which the parents of the new generation are chosen: in the $(\mu + \lambda) - ES$ scheme, new parents are selected from the offspring and old parents; in the $(\mu, \lambda) - ES$ scheme, old parents are not considered.

2.4.3 Evolution with Neural Networks

Evolutionary algorithms, e.g. GA and ES, are suitable for global searching and are easy to implement, and have therefore been combined with other learning methods, including neural network learning. Such hybrid algorithms use a neural network model in applications with *e.g.* prediction, classification and regression, while simultaneously searching, with the evolution algorithm, for a better neural network model (in terms of parameters or topology).

with parameters: Belew *et al.* [76] encouraged in 1990 the combination of genetic algorithms with connectionist learning in neural networks, because "the use of GA provides much greater confidence in the face of the stochastic variation" and also because GA "can allow training times to be reduced by as much as two orders of magnitude". One of the difficulties of this combination lies in the encoding of the connection weights into the discrete and binary genome string. In their paper, real-valued connection weights are assumed to be bounded within a region which will be divided into 2^B intervals where B is application-defined and refers to the length of a substring for each connection weight in the genome string. The real-value of the connection weight can then be transformed to the binary space.

A similar example can be found in [77] where the so-called CMA-ES, an evolution strategy (ES) which adapts the covariance matrix of the mutation distribution, has been applied to the optimization of the weights of neural networks for solving reinforcement learning problems.

with topology: Stanley [78, 79] introduced the "Neuro-Evolution of Augmenting Topologies (NEAT)" method which is able to change the topology of a neural network as well as the connection weights, such that this method can search for a solution in the neural network structure space as well as in the parameter space.

NEAT uses a dynamically expandable representation of structure for encoding, so that it can record not only the structure itself but also changes in the structure. By virtue of this representation, NEAT permits the implementation of mutation and crossover evolutionary operations on the structure of the neural network.

Another important concept of NEAT is the innovation protection. Rather than pushing newly born structures into the arena for competition, NEAT only allows local competition of newly born individuals with individuals of similar structure. This way, novel structures are protected and have time to optimize their structure before being confronted with strong species.

Chapter 3

Spiral Recurrent Neural Networks

Similar to conventional recurrent neural networks, the novel architecture Spiral Recurrent Neural Networks (*SpiralRNN*) [80] has a layered structure with a standard input layer receiving environment data, a hidden layer with self-coupling connections providing temporal memory, and an output layer projecting this temporal information into the target space. This procedure is expressed by the following equations:

$$\begin{aligned}
\text{input} \rightarrow \text{hidden} \quad &: \quad \mathbf{s}^{\dagger} = W_{in}\mathbf{x}_{t-1} + \mathbf{b}_1 && (3.1)\\
\text{hidden} \rightarrow \text{hidden} \quad &: \quad \mathbf{s}_t = \mathscr{H}(W_{hid}\mathbf{s}_{t-1} + \mathbf{s}^{\dagger}) && (3.2)\\
\text{hidden} \rightarrow \text{output} \quad &: \quad \mathbf{x}_t = \mathscr{G}(W_{out}\mathbf{s}_t + \mathbf{b}_2) && (3.3)
\end{aligned}$$

where $\mathscr{H}$ and $\mathscr{G}$ are activation functions of hidden neurons and output neurons respectively, b_1 and b_2 are respectively the bias of hidden neurons and output neurons, $\mathbf{x}_{t-1}$ and $\mathbf{x}_t$ are respectively the network input and output, $\mathbf{s}_t$ and $\mathbf{s}_{t-1}$ are the hidden-state vector at time t and $t-1$ respectively, $\mathbf{s}^{\dagger}$ is the temporal variable representing the hidden-state vector before self-feedback, W_{in}, W_{hid} and W_{out} refer to the corresponding connection-weight matrices from input to hidden layer, within the hidden layer, and from hidden to output layer.

The spiral recurrent neural network (*SpiralRNN*) uses a hidden layer of special structure, such that it can efficiently control the eigenvalue spectrum of the hidden-weight matrix W_{hid}. In the following, the structure of a *SpiralRNN* will be introduced and followed by implementation of *SpiralRNN*s.

3.1 Structure and Eigenvalues

The structure of a *SpiralRNN* is based on "spiral units", which also contribute to the evolutionary operation described in *chapter-5*. Therefore the structure of "spiral units" will be introduced first, followed by the structure of a *SpiralRNN* as an entire architecture and then theoretical explanation on the relationship between the eigenvalue spectrum and network parameter is made. Formulas in the coming text will use the notations listed in *table-3.1*.

Symbol	Descriptions
d	the dimension of data
n_{wei}	the total number of parameters (connection weights) in network
n_{units}	the total number of hidden units
N_{hn}	the total number of hidden neurons in network
n_{hnu}	the number of hidden neurons in (the current) hidden unit
$\{n_{hnu}\}_k$	the number of hidden neurons in the k^{th} hidden unit
$\mathbf{s}_t$	the hidden-state vector at time t
$\mathbf{s}_t^{(k)}$	the hidden-state vector of the k-th hidden unit at time t

Table 3.1: *Terminology of symbols*

3.1.1 Hidden Units

The *SpiralRNN* structure can be broken down into smaller units, namely “hidden units” or “spiral units”. Each hidden unit possesses a group of hidden nodes and synaptic connections between them, as well as connections between the input/output nodes and the enclosed hidden nodes. *Fig-3.1(a)* illustrates a typical hidden unit with three input nodes and three output nodes, where the hidden layer structure is only shown symbolically. Note that hidden nodes are fully connected to all input nodes and all output nodes. More details of the connections inside the hidden layer are shown in *fig-3.1(b)*, where the connections from only one particular neuron to all other neurons in the hidden unit are displayed. With all neurons in the hidden unit aligned clockwise on a circle, values of connection weights are defined such that the connection from one node to its first clockwise neighbor has value β_1, the connection to its second clockwise neighbor has value β_2 and so on. The definition of connection values is applied to all the neurons, so that all connections from neurons to their respective first clockwise neighbors have an identical weight β_1, and all the connections from neurons to their second clockwise neighbors have value β_2, and so on. More detail is provided in *fig-3.1(c)*, which depicts recurrent connections from two hidden neurons. Different combinations of arrow shapes and line styles in *fig-3.1(c)* represent different associated values of connection weights.

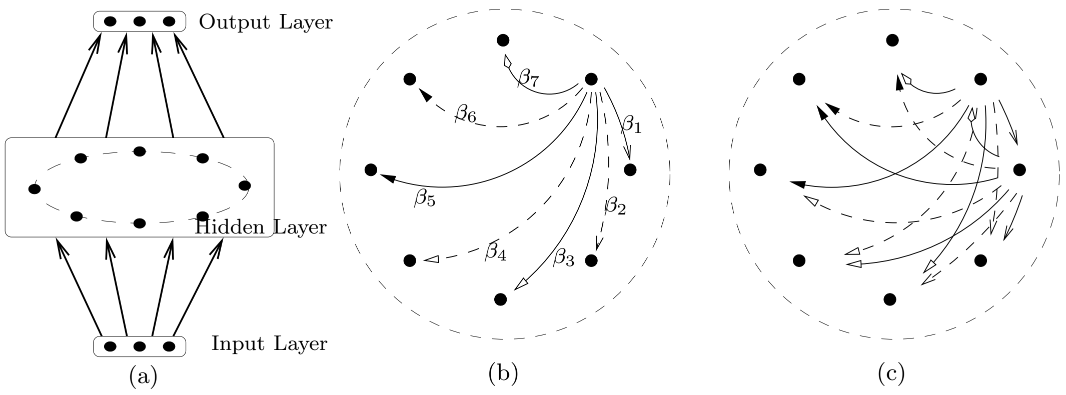


Figure 3.1: *(a) The structure of a hidden unit with 3 input nodes and 3 output nodes; (b) Partial structure of a hidden unit, where only the outgoing connections from one neuron are shown; connections from other neurons will have the same structure of connections and weights. (c) Partial structure of a hidden layer which shows the outgoing connections of two hidden neurons; connections associated with identical values are denoted by the same combination of arrow shape and line style.*

$$M = \begin{pmatrix} 0 & \beta_1 & \beta_2 & \dots & \beta_{n_{hnu}-1} \\ \beta_{n_{hnu}-1} & 0 & \beta_1 & \dots & \vdots \\ \beta_{n_{hnu}-2} & \beta_{n_{hnu}-1} & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \beta_1 \\ \beta_1 & \dots & \dots & \beta_{n_{hnu}-1} & 0 \end{pmatrix}_{n_{hnu}\times n_{hnu}} \qquad \mathcal{P} = \begin{pmatrix} 0 & 1 & & 0 \\ \vdots & \ddots & \ddots & \\ 0 & & \ddots & 1 \\ 1 & 0 & \dots & 0 \end{pmatrix}_{n_{hnu}\times n_{hnu}} \tag{3.4}$$

This configuration of connection weights results in a hidden-weight matrix - matrix M on the left-hand-side of relation *(3.4)*. The value of matrix M is determined by a vector $\vec{\beta} \in \mathbb{R}_{(n_{hnu}-1)\times 1}$ where n_{hnu} refers to the number of hidden neurons in the hidden unit. Furthermore, matrix M can be decomposed into entries of $\vec{\beta}$ and the permutation matrix $\mathcal{P}$:

$$M = \beta_1\mathcal{P} + \beta_2\mathcal{P}^2 + \ldots + \beta_{n_{hnu}-1}\mathcal{P}^{n_{hnu}-1} \tag{3.5}$$

where matrix $\mathcal{P} \in \mathbb{R}_{n_{hnu}\times n_{hnu}}$ shown on the right hand side of relation *(3.4)* is the permutation matrix which up-shifts [3] by one position of entries in a multiplier vector. It is obvious that $\mathcal{P}$'s variant $\mathcal{P}^2$ is also a permutation matrix up-shifting a multiplier vector by two positions. Similarly, $\mathcal{P}^{n_{hnu}}$ up-shifts the multiplier vector by n_{hnu} positions, and

[3] The "up-shift" operation shifts vector entries from the bottom up to the top; For those entries originally at the top, they will be shifted to the bottom circle-wise.

therefore:

$$\mathcal{P} = \mathcal{P}^{n_{hnu}}$$

This also implies that the eigenvalue $\hat{\lambda}_k$ [81] of any permutation matrix $\mathcal{P}^i$ ($i \in \mathbb{N}^+$) satisfies:

$$|\hat{\lambda}_k| = 1, \quad k = 1, \ldots, n_{hnu}$$

Therefore, the maximum absolute eigenvalue of matrix M is bounded, such that the relation in *(3.6)* holds. The proof of the relation in *(3.6)* will be given in *appendix-A.2*.

$$|\lambda_{n_{hnu}}| \leq \sum_{i=1}^{n_{hnu}-1} |\beta_i| \tag{3.6}$$

When the vector $\vec{\beta}$ is defined as the product between a predefined value $\gamma \in \mathbb{R}^+$ (Discussion of γ value will be given in *section-4.6.2.*) and a variable vector $\vec{\xi}$, *i.e.*:

$$\vec{\beta} = \gamma \tanh\left(\vec{\xi}\right), \tag{3.7}$$

matrix M can be rewritten as in the following equation:

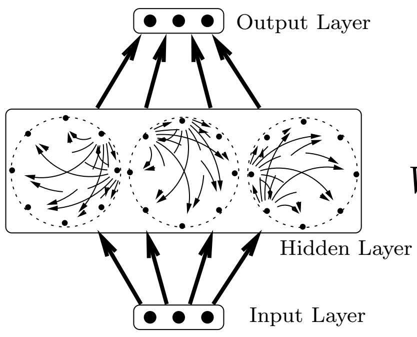
$$M = \sum_{i=1}^{n_{hnu}-1} \gamma \tanh(\xi_i) \mathcal{P}^i,$$

and the relation *(3.6)* can be further developed into relation *(3.8)*.

$$\begin{aligned} |\lambda_{n_{hnu}}| &\leq \gamma \sum_{i=1}^{n_{hnu}-1} |\tanh(\xi_i)| \\ &\leq \gamma(n_{hnu} - 1) \end{aligned} \tag{3.8}$$

3.1.2 ***SpiralRNN*s**

The construction of *SpiralRNN*s is generally based on spiral hidden units. It simply concatenates several hidden units together, and fully connects all hidden neurons to all input and output neurons. Note that hidden units are separated from each other, *i.e.* there is no interconnections between any hidden neuron from one hidden unit and any hidden neuron of another hidden unit (see *fig-3.2(a)*). Assuming the number of hidden units of the entire network is n_{units}, the hidden-weight matrix W_{hid} of the entire network has the form shown in *fig-3.2(b)*. The matrix W_{hid} is a block-diagonal matrix with n_{units} number of blocks, each of which is a square matrix and corresponds to the hidden-weight matrix of one of the hidden units. Note that the sizes of different sub-blocks M_i can differ, one cause for the implementation of the *evolution framework* as presented in *section-5.2*.



(a)

$$W_{hid} = \begin{pmatrix} M_1 & & & \\ & M_2 & & \\ & & \ddots & \\ & & & M_{n_{units}} \end{pmatrix}$$

(b)

Figure 3.2: *(a) The typical structure of* SpiralRNN*s. Note that all hidden units have the same type of topology (however the number of hidden nodes in the hidden units can be different), as shown in fig-3.1, and are separated from each other whereas the input and output connections are fully connected to the hidden nodes. (b) The corresponding hidden-weight matrix W_{hid}, which is a block-diagonal matrix.*

Setting up such a block-diagonal structure helps to establish a constraint upon the eigenvalue spectrum of the hidden-weight matrix W_{hid} as shown in *eq-(3.9)*. The proof will be given in *appendix-A.3*.

$$|\lambda| \leq \max_k \left\{ ||\vec{\beta}^{(k)}||_{taxi} \right\}, \quad k \in [1, \cdots, n_{units}] \tag{3.9}$$

3.1.3 Eigenvalues in *SpiralRNN*s

Besides the boundary condition, parameter vector $\vec{\beta}$ of the synaptic connection inside the hidden-layer is in fact directly connected to the eigenvalues of hidden-weight matrix. According to [82, 83], each sub-block of hidden-weight matrix (ref. *fig-3.2(b)*) is a circulant matrix. Eigenvalue λ_m of circulant matrices is related to entries of the first column $\vec{\beta}$ of matrix, as in *eq-(3.10)*.

$$\lambda_m = \sum_{k=1}^{n} \beta_k e^{-2\pi m(k-1)i/n} \quad m = 1 \dots n \tag{3.10}$$

Here, i is the imaginary unit indicating the complex value and n indicates the matrix size. It is obvious that, λ_m is the discrete Fourier transform (DFT) of the sequence $\vec{\beta}$, whose values are subjected to be learnt during the training of neural network. Therefore, *eq-(3.10)* can be re-written into a linear transformation between eigenvalues $\vec{\lambda}$ and the connection weights $\vec{\beta}$:

$$\vec{\lambda} = \sqrt{n} M_{DFT} \vec{\beta} \tag{3.11}$$

In *eq-(3.11)*, matrix M_{DFT} is the DFT matrix with the form:

$$M_{DFT} = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{n-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & w^{n-1} & w^{2(n-1)} & \cdots & w^{(n-1)(n-1)} \end{bmatrix}_{n\times n} \tag{3.12}$$

where $w = e^{-2\pi i/n}$ is a primitive n-th root of unity. Note that the DFT matrix M_{DFT} is a constant matrix when value of n is fixed. Therefore, during the training of *SpiralRNN* model where value of $\vec{\beta}$ is modified, the eigenvalues $\vec{\lambda}$ of the hidden-weight matrix is adapted according to the linear mapping in *eq-(3.11)*. As such linear mapping is relatively simpler and easier to learn, the training of *SpiralRNN*s can control the eigenvalue-spectrum of hidden-weight matrix in a much more straightforward and efficient manner.

3.2 Implementation of *SpiralRNN*s

Being trained on-line as required by requirement $< \mathfrak{R}1 >$ on *page-3*, the *SpiralRNN* model can be implemented in three modes within one time step (the time interval between the availability of two successive data). They include: the forward phase, the training phase and the autonomous test phase. An implementation in pseudo code is given in *table-3.2*.

The forward phase involves propagation of information from the input layer through the hidden layer to the output layer; The historical information stored in the hidden state melds together with the current environment information, and produces the new hidden-state information, which is further rendered into the network output.

The training phase involves training of network parameters according to the error function; gradient-based methods are fitting by on-line learning without the requirement of large memory; after the adjustment in each time step, the system will continue the forward phase as soon as the input and target data for the next time step are available;

The autonomous test phase of a neural network model is the procedure where the model continuously evolves by following the forward phase, where the network input of the current time step is the output of the previous time step. Note that, throughout this procedure, the model does not interface with data from outside except for the initial input value.

```
for each time step $t$
    obtain the latest data $\hat{x}_t$;

    set the previous data $\hat{x}_{t-1}$ as input of network;        } Forward phase
    iterate the network and generate output $x_t$;

    calculate error: $\delta_t = \hat{x}_t - x_t$;                     } Training phase
    update the connection weights according to the error

    if is required
        set data $\hat{x}_t$ as input                                  } Testing phase
        continuously iterate the network
    end
end
```

Table 3.2: *Pseudo code: implementation of* SpiralRNN*s*

3.2.1 The Forward Phase

For convenience, rewrite the forward equations of *SpiralRNN*s in *eq-(3.1)* to *eq-(3.3)*:

$$\text{input} \rightarrow \text{hidden} \quad : \quad \mathbf{s}^{\dagger} = W_{in}\mathbf{x}_{t-1} + \mathbf{b}_1 \tag{3.13}$$

$$\text{hidden} \rightarrow \text{hidden} \quad : \quad \mathbf{s}_t = \mathscr{H}(W_{hid}\mathbf{s}_{t-1} + \mathbf{s}^{\dagger}) \tag{3.14}$$

$$\text{hidden} \rightarrow \text{output} \quad : \quad \mathbf{x}_t = \mathscr{G}(W_{out}\mathbf{s}_t + \mathbf{b}_2) \tag{3.15}$$

Given the special structure of the hidden-weight matrix W_{hid} in a *SpiralRNN* model described in *section-3.1*, W_{hid} can be converted into a block-diagonal matrix where each block has the form

$$M_k = \sum_{i=1}^{\{n_{hnu}\}_k - 1} \beta_i^{(k)} \mathcal{P}_{(k)}^i, \tag{3.16}$$

where $\{n_{hnu}\}_k$ represents the number of hidden nodes in the k^{th} hidden unit, index $k \in [1, \cdots, n_{units}]$, index $i \in [1, \cdots, \{n_{hnu}\}_k - 1]$, vector $\vec{\beta}^{(k)}$ and matrix $\mathcal{P}_{(k)}$ are respectively the associated vector and the (1-position) permutation matrix of sub-block matrix M_k in W_{hid}. According to the size of sub-block in W_{hid}, vector $\mathbf{s}_t$ can be divided into vectors of smaller size $\mathbf{s}_t^{(k)} \in \mathbb{R}_{\{n_{hnu}\}_k \times 1}$, each $\mathbf{s}_t^{(k)}$ is to the hidden-state vector of the corresponding hidden unit, *i.e.*:

$$W_{hid} = \begin{bmatrix} \ddots & & \\ & \sum_{i=1}^{\{n_{hnu}\}_k - 1} \beta_i^{(k)} \mathcal{P}_{(k)}^i & \\ & & \ddots \end{bmatrix}_{N_{hn} \times N_{hn}}, \quad \mathbf{s}_t = \begin{bmatrix} \vdots \\ \mathbf{s}_t^{(k)} \\ \vdots \end{bmatrix}_{N_{hn} \times 1}, k \in [1, \cdots, n_{units}],$$

where N_{hn} is the total number of hidden nodes. Then, *eq-(3.14)* can be re-written as:

$$\mathbf{s}_t = \mathscr{H}\left(\begin{bmatrix} \vdots \\ \sum_{i=1}^{\{n_{hnu}\}_k-1} \beta_i^{(k)} \mathcal{P}_{(k)}^i \mathbf{s}_{t-1}^{(k)} \\ \vdots \end{bmatrix}_{N_{hn}\times 1} + \mathbf{s}^\dagger\right), \quad k \in [1, \cdots, n_{units}]$$

where the definition of $\mathbf{s}^\dagger$ can be found in *eq-(3.18)*. Furthermore, given the definition $\vec{\beta}^{(k)} = \gamma \tanh(\vec{\xi}^{(k)})$, the equation above will be re-written as:

$$\mathbf{s}_t = \mathscr{H}\left(\gamma \begin{bmatrix} \vdots \\ \sum_{i=1}^{\{n_{hnu}\}_k-1} \tanh\left(\xi_i^{(k)}\right) \mathcal{P}_{(k)}^i \mathbf{s}_{t-1}^{(k)} \\ \vdots \end{bmatrix}_{N_{hn}\times 1} + \mathbf{s}^\dagger\right), \quad k \in [1, \cdots, n_{units}] \tag{3.17}$$

Therefore *eq-(3.13)* to *eq-(3.15)* can be written element-wised, as in *eq-(3.18)* to *eq-(3.20)*. Note that *eq-(3.19)* calculates the hidden-state vector of each hidden unit where vector $s^{\dagger(k)}$ stands for the corresponding sub-vector of $\mathbf{s}^\dagger$.

$$\mathbf{s}^\dagger(i) = \sum_{j=1}^{d} W_{in}(i,j)\mathbf{x}_{t-1}(j) + \mathbf{b}_1(i), \quad i \in [1, \cdots, N_{hn}] \tag{3.18}$$

$$\mathbf{s}_t^{(k)}(i) = \mathscr{H}\left(\gamma \sum_{q=1}^{\{n_{hnu}\}_k-1} \tanh\left(\xi_q^{(k)}\right) \sum_{m=1}^{\{n_{hnu}\}_k} \mathcal{P}_{(k)}^q(i,m) s_{t-1}^{(k)}(m) + s^{\dagger(k)}(i)\right), \quad \text{with} \quad k \in [1, \cdots, n_{units}], i \in [1, \cdots, \{n_{hnu}\}_k] \tag{3.19}$$

$$\mathbf{x}_t(i) = \mathscr{G}\left(\sum_{j=1}^{N_{hn}} W_{out}(i,j)\mathbf{s}_t(j) + \mathbf{b}_2(i)\right), \quad i \in [1, \cdots, d] \tag{3.20}$$

3.2.2 The Training Phase

With the network output readily generated in the forward phase, the training of a *Spiral-RNN* model starts by comparing the output with the target value, *i.e.* determining the error. The error (residual) is put into relation to the change in the network parameters, similar to the gradient-based learning methods. The training follows the extended Kalman filter (EKF) with the gradient calculation based on real time recurrent learning (RTRL). The following will explain the calculation of network output gradient *w.r.t.* all network parameters, then the EKF based algorithm is addressed. A sketch of the training phase is given in *fig-3.3*.

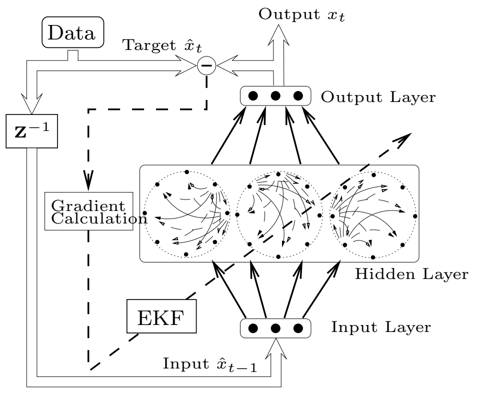


Figure 3.3: *Training of a* SpiralRNN*, with the dashed line representing the learning of network parameters.*

1. *Gradient*

   In order to implement the real time recurrent learning (RTRL) method (refer to *section-2.2.4*), both the output gradient *w.r.t.* parameters $\partial x/\partial w$ and the hidden-state gradient *w.r.t.* parameters $\partial s/\partial w$ are calculated. The $\partial s/\partial w$ will also be saved and recalled in each training step because of the self-coupling in the hidden layer. As the teacher forcing technique (refer to *section-2.2.4*) is used, the gradient calculation is truncated in the sense that the output gradient of the previous time step is not saved. Therefore, at time step t, $\partial x_{t-1}/\partial w = 0$, where x_{t-1} is the input in *eq-(3.13)*.

   In the following, the calculations of $\partial x/\partial w$ and $\partial s/\partial w$ are given in the order of the neural network structure, namely the output layer, the hidden layer and the input layer. Afterward, gradient values $\partial x/\partial w$ and $\partial s/\partial w$ are separately concatenated, and will be further used in the parameter-training algorithm.

   Output layer:

   Eq-(3.20) yields the network output gradient *w.r.t.* the connection weights from the hidden layer to output layer as well as the bias of output nodes. The relevant equations are *eq-(3.21)* and *eq-(3.22)*, with $i, k \in [1, \cdots, d]$ where d is the dimension of the data, $j \in [1, \cdots, N_{hn}]$ where N_{hn} is the total number of hidden neurons, $\mathscr{G}'_k$ is the derivative of the $k-$th activation output *w.r.t.* its corresponding "netin"[4], δ is the Kronecker's delta function. The respective gradient matrix $\partial s/\partial w$ is zero since the hidden-state vector is independent from the weights in the output layer.

[4]The value of "netin" is the neuron value before the activation function is performed, as described in *section-2.1*

$$\frac{\partial x_t(k)}{\partial W_{out}(i,j)} = \mathscr{G}_k' \mathbf{s}_{t(j)} \delta_{i,k} \tag{3.21}$$

$$\frac{\partial x_t(k)}{\partial b_2(i)} = \mathscr{G}_k' \delta_{i,k} \tag{3.22}$$

Hidden layer:

The recurrent connection weights between any two hidden neurons constitute the variables in the hidden layer[5]. These recurrent connections are determined by the associated vectors $\vec{\beta}^{(k)}, k \in [1, \cdots, N_{hn}]$, where the value of $\vec{\beta}^{(k)}$ is determined by the vector variable $\vec{\xi}^{(k)}$ and the predefined $\gamma \in \mathbb{R}^+$. In the following, the network output gradient *w.r.t.* vector $\vec{\xi}$ will be given.

Given the forward equation in *eq-(3.17)*, the derivative of the i^{th} component of vector $s_t^{(k)}$ in the k-th hidden unit *w.r.t.* the j^{th} component of corresponding vector $\vec{\xi}^{(k)}$ is calculated in *eq-(3.23)*, using not only the direct derivative of vector $\vec{\xi}^{(k)}$, but also the corresponding partial derivative from the hidden state vector at the previous time step.

$$\begin{aligned}\frac{\partial s_t^{(k)}(i)}{\partial \xi^{(k)}(j)} &= \mathscr{H}^{(k)\prime}(i) \left(\sum_{m=1}^{\{n_{hnu}\}_k} \mathcal{P}_{(k)}^j(i,m) \mathbf{s}_{t-1}^{(k)}(m) \frac{\partial \beta^{(k)}(j)}{\partial \xi^{(k)}(j)} + \right. \\ &\quad \left. \gamma \sum_{q=1}^{\{n_{hnu}\}_k - 1} \tanh\left(\xi_q^{(k)}\right) \sum_{m=1}^{\{n_{hnu}\}_k} \mathcal{P}_{(k)}^j(i,m) \frac{\partial s_{t-1}^{(k)}(m)}{\partial \xi^{(k)}(j)} \right) \end{aligned} \tag{3.23}$$

$$\text{with} \quad \frac{\partial \beta^{(k)}(j)}{\partial \xi^{(k)}(j)} = \gamma \left(1 - \tanh^2\left(\xi^{(k)}(j)\right)\right) \tag{3.24}$$

where $\mathscr{H}^{(k)\prime}(i)$ is the gradient of $s_t^{(k)}(i)$ *w.r.t.* its corresponding "netin", $\mathcal{P}_{(k)}^j(i,m)$ is the respective element of the permutation matrix $\mathcal{P}_{(k)}^j$ corresponding to the k-th hidden unit, j is the exponential operator, i and m are the indices in the matrix, and $\partial s_{t-1}^{(k)}(i)/\partial \xi^{(k)}(j)$ is the corresponding hidden-state gradient of $s_{t-1}^{(k)}$ *w.r.t.* the j-th entry of vector $\vec{\xi}^{(k)}$.

Note that the hidden-state gradient from one hidden unit *w.r.t.* the ξ value of another hidden unit is always zeros, because hidden units are isolated from each other, *i.e.*:

$$\frac{\partial s_t^{(k1)}(i)}{\partial \xi^{(k2)}(j)} = 0, \quad k1 \neq k2, \quad \forall i, j.$$

[5] The bias of the hidden neurons are like connection weights for an additional constant input in the input layer.

Using the hidden-state gradient, the network output gradient *w.r.t.* corresponding parameters can be computed as followed:

$$\frac{\partial x_t(i)}{\partial \xi^{(k)}(j)} = \mathscr{G}'(i) \sum_{m=1}^{\{n_{hnu}\}_k} W_{out}^{(k)}(i,m) \frac{\partial s_t^{(k)}(m)}{\partial \xi^{(k)}(j)}, \qquad \begin{matrix} i \in [1, \cdots, d] \\ k \in [1, \cdots, n_{units}] \\ j \in [1, \cdots, \{n_{hnu}\}_k] \end{matrix}$$

Input layer:

Variables in the input layer are the connection weights from the input to hidden layer, and the bias of hidden neurons. Similar to *eq-(3.23)*, the derivative of the hidden-neuron activation values *w.r.t.* parameters consists of the direct gradient and the partial derivative:

$$\begin{aligned} \frac{\partial s_t^{(k)}(i)}{\partial W_{in}^{(k)}(m,n)} &= \mathscr{H}^{(k)'}(i) \Bigg(x_{t-1}(n)\delta_{i,n} + \\ &\quad \gamma \sum_{q=1}^{\{n_{hnu}\}_k - 1} \tanh\left(\xi_q^{(k)}\right) \sum_{l=1}^{\{n_{hnu}\}_k} \mathcal{P}_{(k)}^q(i,l) \frac{\partial s_{t-1}^{(k)}(l)}{\partial W_{in}^{(k)}(m,n)} \Bigg) \qquad (3.25) \\ \frac{\partial s_t^{(k)}(i)}{\partial b_1^{(k)}(j)} &= \mathscr{H}^{(k)'}(i) \Bigg(1 + \\ &\quad \gamma \sum_{q=1}^{\{n_{hnu}\}_k - 1} \tanh\left(\xi_q^{(k)}\right) \sum_{l=1}^{\{n_{hnu}\}_k} \mathcal{P}_{(k)}^q(i,l) \frac{\partial s_{t-1}^{(k)}(l)}{\partial b_1^{(k)}(j)} \Bigg) \qquad (3.26) \end{aligned}$$

Furthermore, the derivative of the network output *w.r.t.* respective variables is computed as following:

$$\begin{aligned} \frac{\partial x_t(i)}{\partial W_{in}^{(k)}(m,n)} &= \mathscr{G}'(i) \sum_{l=1}^{\{n_{hnu}\}_k} W_{out}^{(k)}(i,l) \frac{\partial s_t^{(k)}(l)}{\partial W_{in}^{(k)}(m,n)} \\ \frac{\partial x_t(i)}{\partial b_1^{(k)}(j)} &= \mathscr{G}'(i) \sum_{l=1}^{\{n_{hnu}\}_k} W_{out}^{(k)}(i,l) \frac{\partial s_t^{(k)}(l)}{\partial b_1^{(k)}(j)} \end{aligned}$$

2. Training method - extended Kalman filter (*EKF*)

   The training method will be addressed in the following text. First there will be a review of the general scenario of sensor prediction and a definition of the probability distribution function over the variables in the system. The solution for parameter estimation of the learning model is derived based on the probability distribution functions of parameters w given the training data $\hat{\mathbf{x}}_t$, *i.e.* $f_{\mathbf{w}_t|\hat{\mathbb{X}}_t}$, which is the extended Kalman filters (*EKF*) given certain approximations and specifications.

   In order to predict data, each sensor node has to maintain and update a model of the environment allowing the model to predict future observations. A typical starting point for a model of a generic dynamic system is:

$$\mathbf{s}_t = \mathscr{H}\left(\mathbf{s}_{t-1}, \mathbf{w}_t, \hat{\mathbf{x}}_{t-1}\right) \tag{3.27}$$

$$\hat{\mathbf{x}}_t = \mathscr{G}\left(\mathbf{s}_t, \mathbf{w}_t, \hat{\mathbf{x}}_{t-1}\right) + \nu_t \tag{3.28}$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \mu_t \tag{3.29}$$

Eq-(3.27) describes the dynamic evolution of the environment hidden-state vector, *i.e.* the new hidden state vector $\mathbf{s}_t$ at time t. The value of $\mathbf{s}_t$ depends in some, usually nonlinear, way on the previous hidden state vector $\mathbf{s}_{t-1}$ and the previous observation vector $\hat{\mathbf{x}}_{t-1}$ and the model parameters $\mathbf{w}_t$. *Eq-(3.28)* describes the measurement process of sensor: $\hat{\mathbf{x}}_t$ is the vector of observations (sensor data) at time t, $\mathscr{G}(\dots)$ is the model estimation of these observations based on $\mathbf{s}_t$, $\mathbf{w}_t$ and $\hat{\mathbf{x}}_{t-1}$. The variable ν_t is the measurement noise vector, assumed to be normally distributed with a probability distribution function (*p.d.f.*) $f_{\nu_t} = \mathscr{N}(\nu_t \,|0, R_t)$. Finally, *eq-(3.29)* describes the evolution of the model parameters: that the dynamics of the environment within a reasonable time window are assumed to be stationary, *i.e.* the model parameters are static, up to additive random fluctuations μ_t; these are assumed to be normally distributed with zero mean, covariance matrix Q_t and thus with a corresponding *p.d.f.* $f_{\mu_t} = \mathscr{N}(\mu_t \,|0, Q_t)$.

Using the Bayes rule, the chain rule and the Chapman-Kolmogorov equation [56], the conditional *p.d.f.* $f_{\mathbf{w}_t|\hat{\mathbb{X}}_t}$ for the model parameters at time t given the current and all previous observations $\hat{\mathbb{X}}_t = \{\hat{\mathbf{x}}_t, \hat{\mathbf{x}}_{t-1}, ..., \hat{\mathbf{x}}_0\}$ is given by following equations:

$$f_{\mathbf{w}_t|\hat{\mathbb{X}}_t} = f_{\mathbf{w}_t|\hat{\mathbf{x}}_t,\hat{\mathbb{X}}_{t-1}} = \frac{f_{\hat{\mathbf{x}}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} f_{\mathbf{w}_t|\hat{\mathbb{X}}_{t-1}}}{f_{\hat{\mathbf{x}}_t|\hat{\mathbb{X}}_{t-1}}} \tag{3.30}$$

$$f_{\hat{\mathbf{x}}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} = \int f_{\hat{\mathbf{x}}_t|\mathbf{s}_t\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} f_{\mathbf{s}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} d\mathbf{s}_t \tag{3.31}$$

$$f_{\mathbf{w}_t|\hat{\mathbb{X}}_{t-1}} = \int f_{\mathbf{w}_t|\mathbf{w}_{t-1},\hat{\mathbb{X}}_{t-1}} f_{\mathbf{w}_{t-1}|\hat{\mathbb{X}}_{t-1}} d\mathbf{w}_{t-1} \tag{3.32}$$

Eq-(3.30) is the famous Bayes rule, *eq-(3.31)* introduces the hidden-state vector $\mathbf{s}_t$ into the "evidence" *p.d.f.* and *eq-(3.32)* is the "prior" *p.d.f.* for the model parameters $\mathbf{w}_t$. The last equation introduces the conditional *p.d.f.* $f_{\mathbf{w}_{t-1}|\hat{\mathbb{X}}_{t-1}}$ which allows to interpret *eq-(3.30)* to *eq-(3.32)* as iterative update equations for $f_{\mathbf{w}_t|\hat{\mathbb{X}}_t}$.

Before the on-line training rules from these update equations are derived, one has to introduce some assumptions and approximations, and they present in the *specification* of each *p.d.f.*:

($\mathbf{S}_a$) $f_{\mathbf{w}_{t-1}|\hat{\mathbb{X}}_{t-1}} \approx \mathscr{N}(\mathbf{w}_{t-1} \,|\bar{\mathbf{w}}_{t-1}, P_{t-1})$ is assumed to be normally distributed with mean $\bar{\mathbf{w}}_{t-1}$ and covariance matrix P_{t-1}. In general this *p.d.f.* will be multi-modal; however, if the environment doesn't change its dynamics too fast and if the model is powerful enough, then the model parameters should become static and this assumption is justified.

($\mathbf{S}_b$) $f_{\mathbf{w}_t|\mathbf{w}_{t-1},\hat{\mathbb{X}}_{t-1}} = \mathscr{N}(\mathbf{w}_t\,|\mathbf{w}_{t-1}, Q_t)$ according to the assumption made in *eq-(3.29)*.

($\mathbf{S}_c$) $f_{\mathbf{s}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} = \delta(\mathbf{s}_t - \mathscr{H}(\mathscr{H}(\ldots,\mathbf{w}_t,\hat{\mathbf{x}}_{t-2}),\mathbf{w}_t,\hat{\mathbf{x}}_{t-1}))$ with the dots indicating an iteration of the function $\mathscr{H}(\ )$ which further represents the previous hidden states $\mathbf{s}_{t-\tau}(\tau = 1, 2, \ldots)$ according to *eq-(3.27)*. The Dirac delta-function $\delta(\ )$ corresponds to a normal distribution with infinitesimally small covariance matrix and reflects the assumption that any uncertainty comes from the measurements and from (random) changes of model parameters.

($\mathbf{S}_d$) $f_{\hat{\mathbf{x}}_t|\mathbf{s}_t\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}}$ in *eq-(3.31)* is normal distributed with mean value $\mathscr{G}(\mathbf{s}_t,\mathbf{w}_t,\hat{\mathbf{x}}_{t-1})$ and covariance matrix R_t according to *eq-(3.28)*, *i.e.* $f_{\hat{\mathbf{x}}_t|\mathbf{s}_t\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} = \mathscr{N}(\hat{\mathbf{x}}_t\,|\mathscr{G}(\mathbf{s}_t,\mathbf{w}_t,\hat{\mathbf{x}}_{t-1}), R_t)$.

With the specifications ($\mathbf{S}_a$) and ($\mathbf{S}_b$), *eq-(3.32)* is a convolution of normal distributions and thus $f_{\mathbf{w}_t|\hat{\mathbb{X}}_{t-1}}$ can be easily calculated:

$$f_{\mathbf{w}_t|\hat{\mathbb{X}}_{t-1}} = \mathscr{N}\left(\mathbf{w}_t\left|\bar{\mathbf{w}}_{t-1}, P_t^\dagger\right.\right) \quad \textit{with} \quad P_t^\dagger = P_{t-1} + Q_t \tag{3.33}$$

Given the δ-function like *p.d.f.* $f_{\mathbf{s}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}}$ in specification ($\mathbf{S}_c$) and the *p.d.f.* of $f_{\hat{\mathbf{x}}_t|\mathbf{s}_t\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}}$ in specification ($\mathbf{S}_d$), one finds:

$$f_{\hat{\mathbf{x}}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} = \mathscr{N}\left(\hat{\mathbf{x}}_t\,|\mathscr{G}\left(\mathbf{s}_t,\mathbf{w}_t,\hat{\mathbf{x}}_{t-1}\right), R_t\right) \tag{3.34}$$

$$\mathbf{s}_t = \mathscr{H}\left(\mathscr{H}\left(\ldots,\mathbf{w}_t,\hat{\mathbf{x}}_{t-2}\right),\mathbf{w}_t,\hat{\mathbf{x}}_{t-1}\right) \tag{3.35}$$

The same assumption as made in specification ($\mathbf{S}_a$) for $f_{\mathbf{w}_{t-1}|\hat{\mathbb{X}}_{t-1}}$ - namely the slow change of the model parameters $\mathbf{w}_t$ when the model has been well trained - justifies another approximation: the function $\mathscr{G}(\ldots)$ can be linearized with respect to their dependence on the model parameters $\mathbf{w}_t$. In particular,

$$\mathscr{G}(\mathbf{s}_t,\mathbf{w}_t,\hat{\mathbf{x}}_{t-1}) \approx \mathbf{x}_t + \Psi(\mathbf{w}_t - \bar{\mathbf{w}}_{t-1}), \tag{3.36}$$

where the output vector $\mathbf{x}_t$ and the gradient matrix Ψ are respectively defined as:

$$\mathbf{x}_t = \mathscr{G}\left(\mathbf{s}_t^\dagger,\bar{\mathbf{w}}_{t-1},\hat{\mathbf{x}}_{t-1}\right) \quad \text{and} \quad \Psi = \frac{d\mathbf{x}_t}{d\bar{\mathbf{w}}_{t-1}} \tag{3.37}$$

Finally, the definition of $\mathbf{s}_t^\dagger$ is given here. According to *eq-(3.35)* and *eq-(3.36)*, $\mathbf{s}_t^\dagger$ should be the hidden state vector calculated with the previous model parameters $\bar{\mathbf{w}}_{t-1}$. However, this requires a recalculation of the whole history whenever the model parameters change and therefore is not a viable solution for on-line learning in distributive sensor application. Instead, the approximation in *eq-(3.38)* is applied, which implies that the model takes $\mathbf{s}_t^\dagger$ as the flawless hidden value.

$$\mathbf{s}_t^\dagger \approx \mathscr{H}\left(\mathscr{H}\left(\ldots,\bar{\mathbf{w}}_{t-2},\hat{\mathbf{x}}_{t-2}\right),\bar{\mathbf{w}}_{t-1},\hat{\mathbf{x}}_{t-1}\right) \tag{3.38}$$

With the approximations in *eq-(3.36)*, *eq-(3.37)* and *eq-(3.38)*, the calculation of $f_{\hat{\mathbf{x}}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}}$ in *eq-(3.34)* is rewritten as:

$$f_{\hat{\mathbf{x}}_t|\mathbf{w}_t,\hat{\mathbb{X}}_{t-1}} = \mathscr{N}\left(\hat{\mathbf{x}}_t \middle| \mathscr{G}\left(\mathbf{s}_t^\dagger, \bar{\mathbf{w}}_{t-1}, \hat{\mathbf{x}}_{t-1}\right) + \Psi(\mathbf{w}_t - \bar{\mathbf{w}}_{t-1}), R_t\ .\right) \tag{3.39}$$

Combining *eq-(3.33)* and *eq-(3.39)*, the *p.d.f.* $f_{\mathbf{w}_t|\hat{\mathbb{X}}_t}$ in *eq-(3.30)* ends up with a normal distribution in *eq-(3.40)*. Note that $P_t^\dagger$ is computed in *eq-(3.33)* and Ψ is the gradient described on *page-35*.

$$\begin{aligned} f_{\mathbf{w}_t|\hat{\mathbb{X}}_t} &= \mathscr{N}(\mathbf{w}_t\,|\bar{\mathbf{w}}_t, P_t) & (3.40)\\ \text{with}\quad \bar{\mathbf{w}}_t &= \bar{\mathbf{w}}_{t-1} + P_t\Psi^T R_t^{-1}(\hat{\mathbf{x}}_t - \mathbf{x}_t) & (3.41)\\ P_t &= P_t^\dagger - P_t^\dagger\Psi^T\left(\Psi P_t^\dagger\Psi^T + R_t\right)^{-1}\Psi P_t^\dagger & (3.42)\end{aligned}$$

Eq-(3.40) is consistent with specification ($\mathbf{S}_a$) whilst *eq-(3.41)* and *eq-(3.42)* constitute the extended Kalman filter (EKF) (for more details see [55, 56, 57]).

The derivation of the on-line learning rule using a Bayesian approach implies that this rule is optimal, provided that the above specifications and approximations are justified - optimal in the sense that the learning rule yields the most likely model parameters at time t given a history of observations made so far.

The choice and update of the parameter matrices Q_t and R_t, which are respectively the covariance matrices of noise nu_t and mu_t, will be described in simulation setting in *section-4.1.3*. Calculation of gradient Ψ in *eq-(3.37)* can be generally expressed in *eq-(3.43)* (for more details on gradient calculation, please refer to previous text on *page-35*). Note that *eq-(3.43)* is calculated in an iterative manner with the approximation in *eq-(3.38)*.

$$\begin{aligned}\Psi &= \frac{d\mathbf{x}_t}{d\bar{\mathbf{w}}_{t-1}} = \frac{\partial\mathbf{x}_t}{\partial\bar{\mathbf{s}}_t^\dagger}\frac{d\mathbf{s}_t^\dagger}{d\bar{\mathbf{w}}_{t-1}} + \frac{\partial\mathbf{x}_t}{\partial\bar{\mathbf{w}}_{t-1}}\\ &= \frac{\partial\mathscr{G}\left(\mathbf{s}_t^\dagger, \bar{\mathbf{w}}_{t-1}, \hat{\mathbf{x}}_{t-1}\right)}{\partial\mathbf{s}_t^\dagger}\frac{d\mathbf{s}_t^\dagger}{d\bar{\mathbf{w}}_{t-1}} + \frac{\partial\mathscr{G}\left(\mathbf{s}_t^\dagger, \bar{\mathbf{w}}_{t-1}, \hat{\mathbf{x}}_{t-1}\right)}{\partial\bar{\mathbf{w}}_{t-1}} & (3.43)\\ \text{with}\quad \frac{d\mathbf{s}_t^\dagger}{d\bar{\mathbf{w}}_{t-1}} &= \frac{\partial s_t^\dagger}{\partial\bar{\mathbf{w}}_{t-1}} + \frac{\partial s_t^\dagger}{\partial s_{t-1}^\dagger}\frac{ds_{t-1}^\dagger}{d\bar{\mathbf{w}}_{t-1}} \simeq \frac{\partial s_t^\dagger}{\partial\bar{\mathbf{w}}_{t-1}} + \frac{\partial s_t^\dagger}{\partial s_{t-1}^\dagger}\frac{ds_{t-1}^\dagger}{d\bar{\mathbf{w}}_{t-2}} & (3.44)\end{aligned}$$

The latter expression in *eq-(3.44)* is the gradient calculated in real time recurrent learning (RTRL) [66] with teacher forcing. Potential problems may occur, if some eigenvalues of the Jacobian matrix $\frac{\partial s_t^\dagger}{\partial s_{t-1}^\dagger}$ are too large; this may lead to an unlimited amplification of gradient components.

From above, an RTRL-gradient based *EKF* learning rule has been derived, based on a Bayesian approach and with reasonable approximations. The resulting maximum likelihood estimation of the model parameters guarantees the convergence of the learning process - at least for an environment whose dynamics do not change. Different from the adaptive learning rule in [84], also aiming at a fix-point dynamics for the model parameters, this learning rule is derived from a probabilistic approach. In the next chapter, this learning rule together with gradient calculated by teacher forcing based RTRL is taken as the generic on-line learning rule for all neural network models.

3.2.3 The Autonomous Test Phase

The *autonomous test* refers to the test procedure, where the system operates monologically and generates the autonomous output. Autonomous output refers to a network output sequence generated within one autonomous test. Assuming that a model f has been trained on-line with the training data $\{\hat{x}_1, \ldots, \hat{x}_t\}$, the autonomous test of length τ is initialized by giving $\hat{x}_t$ as the model input and iterates the trained model τ times, where in each time step the network input takes the output value at previous step time, as shown in *eq-(3.45)*. The output x_{t+k} is the k-step ahead prediction output based on initial value $\hat{x}_t$. The autonomous output is then constructed by concatenating all prediction outputs of the autonomous test: $\{x_{t+1}, \ldots, x_{t+\tau}\}$, where its corresponding target is $\hat{\{}x_{t+1}, \ldots, \hat{x}_{t+\tau}\}$.

$$\begin{aligned} x_{t+k+1} &= f(x_{t+k}), \quad k = 0, \ldots, \tau - 1 && (3.45) \\ \text{with} \quad x_t &= \hat{x}_t && (3.46) \end{aligned}$$

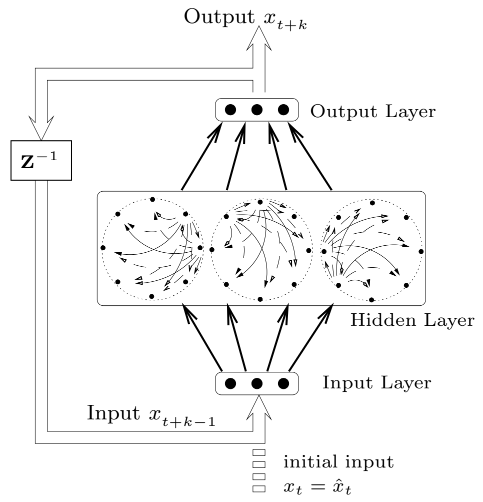


Figure 3.4: *Autonomous test of* SpiralRNN*s, with the dashed line at the initial input indicating that no further data will serve as input data.*

In simulations, *autonomous tests* will be conducted at assigned time steps, namely test points (ref. *section-4.1.4*), after a training phase. Before starting the *autonomous test*, environment data will be supplied as feed-in values for the input layer of the *SpiralRNN* model. After that, the structure and connection weights will be held unchanged, meanwhile the model iterates autonomously (as described in *section-3.2.1*) and generates the autonomous output. The iteration of the model will be terminated when the autonomous output vector has reached to a predefined length. *Fig-3.4* illustrates the autonomous test procedure of the *SpiralRNN*.

Chapter 4

Applications with *SpiralRNN*s

In this chapter, the spiral recurrent neural network (*SpiralRNN*) structure will be compared with state-of-art recurrent neural network (RNN) models in various simulations. These simulations include the prediction on benchmark time series, experiments with real-world data and conditional prediction simulations, as well as the experience in a data prediction competition. Before explaining the detail of each comparison, the experimental setup will be described, as all experiments share a largely common network settings.

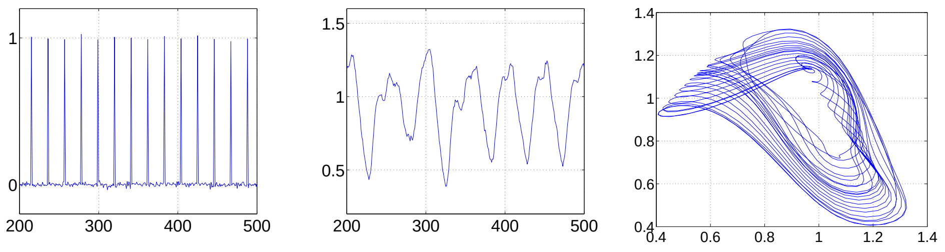
Due to scarce energy and memory of sensor nodes, training of neural networks is performed in an on-line manner, where in each time step the network parameters (*i.e.* connection weights) are updated after comparing the network output to the target data value, as this is required in sensor network applications where no large memory is available. Since energy consumption is a major concern in sensor network applications, the experiments are compared under the condition that the processing effort for training of other networks is roughly on the same level. For statistics reasons, each task task is performed 30 times in order to achieve accuracy and confidence on the result.

Simulations are carried out in the MatLab® environment. The MatLab scripts implementing *SpiralRNN* are given in *appendix-B*.

4.1 Experimental Settings

4.1.1 Tasks

In prediction comparisons, several data sets are used, including the Lorenz time series data set, the spike time series data set and the Mackey-Glass data set. Chaotic time series are used here because they never return to a previous data value and thus probe the models' ability of generalization. Spike time series contain long sequences of null-activity which requires a certain memory horizon in order to predict the spikes. All of these time series are disturbed by additive noise of normal distribution[6], *i.e.*: $\eta_t \sim \mathcal{N}(0, 0.01^2)$. *Fig-4.1* depicts some examples of these time series.



(a) *Spike21* (b) *Mackey-Glass* (c) *Mackey-Glass Orbit*

Figure 4.1: *Examples of time series, where the X-axis is time and the Y-axis is data. (a) Example of* Spike21*; (b) example of* Mackey-Glass*; (c) example of* Mackey-Glass *periodical orbit, x_t versus $x_{t-\tau}$ with $\tau = 17$.*

1. One-dimensional spike time series are used for testing stability and memory horizon. In the simulations, the spike time series has the period 21, *i.e.* in each period one single entry has value 1 whilst all other 20 entries zero. This task and the data set is named *Spike21*, because of its period length. The variance of *Spike21* is manually set to 1, *i.e.* $\sigma^2 = 1$, where the value of variance will be required for the performance evaluation. An example of the data can be seen in *fig-4.1(a)*

2. The Mackey-Glass chaotic time series, proposed in [85] as a model for the production of white blood cells, satisfies the differential equation in *eq-(4.1)*

$$\dot{x}_{(t)} = \frac{a x_{t-\tau}}{1 + x^c_{t-\tau}} - b x_t, \quad t \geq 30 \tag{4.1}$$

   where parameters take on the settings: $a = 0.2$, $b = 0.1$, $c = 10$, $\tau = 17$. Data x_t at the beginning will be initialized with the value of 0.5, whilst from $t \geq 30$ the data value is obtained by applying *eq-(4.1)* with step size $h = 1$. Only data with $t \geq 50$ are used for training and evaluation. in order to avoid the initial part. An example of one-dimensional Mackey-Glass data is given in *fig-4.1(b)*; *fig-4.1(c)* demonstrates the periodical aspect of the chaotic data set. The variance of Mackey-Glass data reads: $\sigma^2 = 0.0518$.

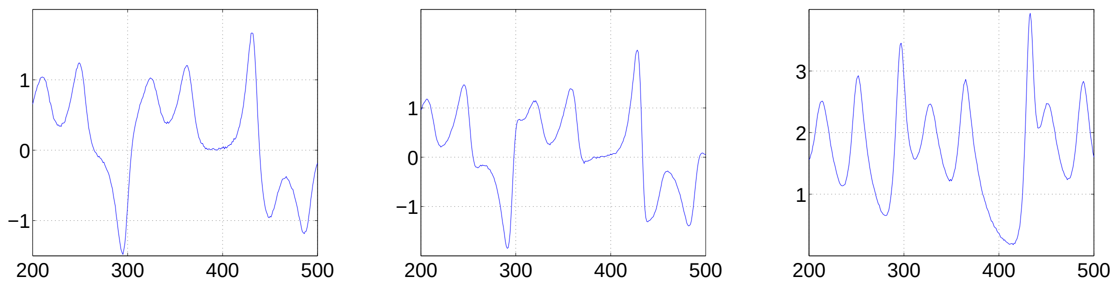
3. As a benchmark chaotic time series, the Lorenz time series has been frequently employed for stability tests because of its sensitivity to the initial conditions. Learning

[6] The normal distribution of a variable v is denoted by $\mathcal{N}(\bar{v}, \sigma^2)$, with $\bar{v}$ being the mean of variable and σ^2 being the variance. The same denotation will be used in the following text.

the dynamics driven by the *Lorenz* data set can be a challenge for neural network models due to the complexity of dynamics involved, which can be described by a three-dimensional chaotic system:

$$\begin{aligned} \dot{x} &= 16(y - x) \\ \dot{y} &= (40 - z)x - y \\ \dot{z} &= xy - 6z \end{aligned} \tag{4.2}$$

Euler's method is used for the integration of the time series with step size $\Delta h = 0.01$ and the initialization $x_0 = 0.1; y_0 = 0.1; z_0 = -0.1$. The time series are scaled down by a factor of 20 such that its value is bounded within a reasonable range. An example of Lorenz data can be found in *fig-4.2*. The variance of the 3-dimension Lorenz time series reads: [0.5473,0.7298,0.5251].



(a) 1st (b) 2nd (c) 3rd

Figure 4.2: *Examples of the Lorenz chaotic time series, where the X-axis is time and the Y-axis is data of the respective dimension. (a)* 1st *dimension; (b)* 2nd *dimension; (c)* 3rd *dimension;*

4.1.2 Networks

The computational complexity in the extended Kalman filter (EKF) depends on the number of model parameters (synaptic weights and biases) to be optimized. Since the effort spent on EKF computation dominates the computational cost of on-line training, for fairness reasons, comparisons are implemented under the condition that all network models have a similar number of parameters to be optimized. In the following, this number of parameters is called the complexity of the system, or simply *complexity*, and is denoted by $\mathscr{C}_s$.

Given the value of $\mathscr{C}_s$ and the dimensionality d of the time series, one can determine the size and topology of the different networks. Including bias on hidden and output neurons[7], the

[7] The bias is not included in the echo state networks (*ESN*), see [51]

total number of trainable parameters in the *SpiralRNN* model, for example, is calculated as $\mathscr{C}_s = 2N_{hn}(d+1) + d$, with N_{hn} referring to the amount of hidden nodes and d referring to the data dimension, as indicated in *table-4.1.2*. Italic values in *table-4.1.2* indicate the total number of trainable connection weights, and the number of hidden neurons in each network when $\mathscr{C}_s$ is roughly[8] equal to expected values of 50, 100 and 200 respectively.

d	(exp.) $\mathscr{C}_s$	actual $\mathscr{C}_s$ / (number of hidden nodes) N_{hn}			
dim. of data	exp. complexity of system $\mathscr{C}_s$	*SpiralRNN* $2N_{hn}(d+1)+d$	*ESN* $(N_{hn}+d)d$	*BDRNN* $2N_{hn}(1+d)+d$	*SRN* ${N_{hn}}^2+2dN_{hn}+N_{hn}+d$
1	50	*60/15*	*61/60*	*57/14*	*55/6*
	100	*100/25*	*111/110*	*105/26*	*109/9*
2	100	*120/20*	*124/60*	*122/20*	*128/9*
	200	*192/32*	*204/100*	*206/34*	*206/12*
3	100	*96/12*	*129/40*	*115/14*	*101/7*
	200	*192/24*	*219/70*	*211/26*	*201/11*

Table 4.1: *The actual network size and number of hidden nodes for each RNN structure in different scenarios (dimension of data, network complexity). The first column states the dimensionality (from 1 to 3) of each tasks. The second column is the expected network size, ranging from 50 to 200. The rest of the columns itemizes the actual network size and the number of hidden nodes, separating by "/". The formulas below the model names indicate how to calculate the actual total number of parameters given the values of d and* N_{hn}.

Without explicit notification, the hyperbolic tangent "tanh" serves as the hidden activation function of each network model, and the output activation function is the identity function. By default, values of all the connection weights are initialized such that they satisfy the normal distribution $\mathscr{N}(0, 0.01^2)$.

Construction of the *SpiralRNN* model follows the description in *chapter-3*, with $\gamma = 1$ for all simulations and experiments.

For the *ESN* model, according to [50], the hidden weight matrix W_{hid} is uniformly randomly initialized over the range $[-1, 1]$ with 95% sparsity, and then the matrix is rescaled such that the maximum eigenvalue λ_{max} is equal to 0.80 in order to set the spectral radius. All the other connection weights in the *ESN* model have uniform random initial values which vary over the range $[-0.1, 0.1]$; the connections include those from input to hidden layer, those from input and hidden layer to output layer, and those from output layer to hidden layer[9]. Values of all connection weights are fixed, except for the connections from input

[8] It is not always possible to construct a neural network whose complexity $\mathscr{C}_s$ exactly equals an arbitrary value.

[9] Even though connections from output to hidden layer are optional and are not included in [50], here they are nevertheless included in the *ESN* model in order to have better performance.

and hidden layer to output layer.

To construct the scaled orthogonal version of the *BDRNN* model (for details please refer to *chapter-2* on *page-16*), for each block matrix $\begin{bmatrix} w_1 & w_2 \\ -w_2 & w_1 \end{bmatrix}$, a parameter-pair $\{w_1, w_2\}$ is defined as:

$$w_1 = \Upsilon sin(\Theta),\ w_2 = \Upsilon cos(\Theta)$$

where $\Upsilon = \tanh(\Phi) \in (-1, 1)$ and $\Phi \in \mathbb{R}$. With such parametrization, it is satisfied that:

$$\begin{aligned} w_1^2 + w_2^2 &= (\Upsilon sin(\Theta))^2 + (\Upsilon cos(\Theta))^2 \\ &= \Upsilon^2 = (\tanh(\Phi))^2 \\ &\leq 1 \end{aligned}$$

Therefore, training of w_1, w_2 has become the problem of training Φ, Θ. The gradient calculation has to be adjusted accordingly.

4.1.3 Training

All neural network models are trained in an on-line manner (refer to *section-2.2*), which means that the training and the update of network parameters occur at each time step when a new target value is given. During on-line training, for example at time step t, input[10] $\hat{x}_{t-1}$ is fed into the input layer of the neural network, the network iterates thereafter with updating the hidden state and generates the output x_{t+1}, which is then compared to the target data $\hat{x}_t$. Model parameters (weights) will be modified according to the error and the training algorithm. This concludes the on-line training at time step t.

The on-line training keeps running in each time step, including the time steps with autonomous tests. For more information about autonomous testing and test points, please refer to the next section on *section-4.1.4*. Training will stop if the absolute value of the gradient has been accumulated so high such that MatLab complains and returns warning messages. However, this has only been observed in simulations with the *SRN* model.

Training of networks has been implemented with teacher forcing RTRL-based EKF model, except for *ESN* models where the recurrent weights are not trained, thus no RTRL algorithm is implemented.

About the parametrization of EKF method, the covariance matrix P_t is initialized as the identity matrix $\mathbf{I_d}$ of a size dependent on the model complexity $\mathscr{C}_s$. The identity matrix implies that at the beginning there is large uncertainty in the model parameters but no *a*

[10] Due to teacher forcing, data $\hat{x}_{t-1}$ is used rather than the previous output x_{t-1}.

priori correlation between them. The model noise covariance matrix Q_t is constant and is set to a diagonal matrix whose diagonal entries are all equal to 10^{-8}, *i.e.* $Q_t = 10^{-8}\mathbf{I_d}$, thus allowing a small fluctuation of the model parameters in order to adapt to the switching of data dynamics. The measurement covariance matrix R_t is initialized as $10^{-2}\mathbf{I_d}$, but its value can be on-line adjusted according to the output error $\mathbf{e} = \mathbf{x}_t - \hat{\mathbf{x}}_t$. This is done using the moving-average technique with a coefficient $\alpha = 0.01$ in *eq-(4.3)*. The employment of fluctuation variable η is for the sake of preventing the singularity of matrix R_t. Practically, the addition of $\eta\mathbf{I_d}$ is not necessary because of the variousness of $\mathbf{e}$ value. The value of η is set to $\eta = 0$ in the simulations.

$$R_t = (1-\alpha)R_{t-1} + \alpha\left(\mathbf{e}\cdot\mathbf{e}^T + \eta\mathbf{I_d}\right) \tag{4.3}$$

4.1.4 Testing and Measurement

Autonomous testing starts when the accumulated time step t coincides with any of the predefined test points. Different comparison tasks are scheduled for different test points. For example, in the task of *Spike21*, test points [11] are set for the time step $t = \{10^{[3:0.5:5]}\}$.

An autonomous test takes the initial network input and applies the update equations without any other data, where at each time step the network input value takes on the value of the network output from the previous time step (refer to *section-3.2.3*). The number of autonomous iterations depends on the particular task, varying from 200 to 2000 time steps. During the whole autonomous test, the values of the network connections are held constant. At the end of the autonomous test, the autonomous output is the concatenation of the network outputs of all autonomous time steps. After the autonomous test, the hidden-state vector will be restored to the original value right before the beginning of the autonomous test.

The measurement of prediction performance is based on the logarithmic normalized mean square error (*logNMSE*). The value of *logNMSE* is based on the comparison between the successive autonomous output vector $\vec{x}$ and the respective target vector $\hat{\vec{x}}$. Assuming the length of autonomous output is $\tilde{\imath}$, the data variance is given as σ^2, it follows that:

$$\begin{aligned} \tilde{\varepsilon} &= \frac{1}{\tilde{\imath}}\sum_{t=1}^{\tilde{\imath}} \log_{10}\left((x_t - \hat{x}_t)^2/\sigma^2\right) \\ \varepsilon &= \frac{1}{d}\sum^{d}\tilde{\varepsilon} \end{aligned} \tag{4.4}$$

[11] Expression [3 : 0.5 : 5] uses the MatLab colon operator ":" and it results in a vector with the first and last entries equal to 3 and 5 respectively, the difference between any two successive entries being 0.5. The list of test points in this case is $[1000, 3162, 10000, 31623, 100000]$.

where the values of variance σ^2 of each data set can be found in *section-4.1.1*. In the case of multi-dimensional data, the value of ε will be the mean over the dimensions, as shown in above equation. The final result will be the mean value over the 30 runs of simulations for all tasks. The standard deviation of the *logNMSE* will be provided to investigate the stability of the result.

In the calculation of *logNMSE* in *eq-(4.4)*, the logarithm is applied before the arithmetic mean, thus yielding a result which is closer to the geometric mean value of the normalized error. By doing this, *logNMSE* has weighted the error in favour of short-term prediction rather than long-term prediction. This corresponds to the fact that, in general, an error in a short-term prediction is exponentially propagated to the long-term prediction, thus making the prediction worthless. Therefore it makes sense to concentrate on the short-term prediction when a general criterion for error assessment is required.

4.2 Simulations with Time Series Prediction

As mentioned in the previous section, simulations of each scenario are performed 30 times where autonomous outputs are generated at each assigned test point in each simulation. Assessment is based on the *logNMSE* values of these autonomous outputs.

Result tables in this section are organized as follows: the first column shows a running number of the test points assigned to the task, simulation results are reported in the remaining columns, grouped in pairs for each RNN model. Inside each pair of columns, the left column lists the average of *logNMSE* over 30 runs and the right column lists the associated standard deviation value. In some experiments with the *SRN* model, simulations stopped half-way due to instability. In such cases, the resulting mean value of *logNMSE* will be quoted in parentheses, and they are calculated without taking those canceled simulation runs into account. The corresponding value in the "*std.*" column states the number of invalid runs in parentheses.

In diagrams showing *logNMSE* results, the X-axis indicates test points and the Y-axis stands for the *logNMSE* evaluation value. Note that the *logNMSE* value itself is on a logarithmic scale. In some tasks, some RNN models have performed badly such that their mean *logNMSE* values differed much from the others; these values are therefore omitted for the clarity of the diagram. Vertical bars at the resulting data points in *logNMSE* diagrams represent standard deviations of the corresponding RNN model. For clarity's sake, only the standard deviation of the *SpiralRNN* model is shown in such diagrams. The standard deviation values of the other RNN models can be found in respective tables.

4.2.1 *Spike21* Dataset

Simulations with the *Spike21* data have been conducted twice, where the network complexity $\mathscr{C}_s$ was set to 50 in the first test and to 100 in the second. In both cases, five test points for autonomous test were placed at time steps $t = \{10^{[3:0.5:5]}\}$ and each autonomous test lasted for $\tilde{\iota} = 2000$ time steps, *i.e.* models were forced to iterate for 2000 steps without any interference from outside!

A specialty of the *Spike21* data set lies in its values being constant (constant "0" in this case) for most of the time, which means that the data presented to the input layer are identical at most of the time. Therefore, a standard *TDNN* model, with limited knowledge about the problem, could have a smaller number of history data patterns in the input layer than the period length of the time series in the equation, and thus would fails to predict the arrival of spikes because it would lack information about their period. In such a case, *TDNN* will try to minimize the output error by searching the cluster center of all the data, which equals to $\frac{1}{21} \simeq 0.05$ in the *Spike21* case. Such a situation can also occur in conventional recurrent neural networks, particularly at the beginning when they are not yet well trained. However, even in such a case with bad performance, conventional RNN models can still yield promising *logNMSE* evaluation results because of the closeness between value "$\frac{1}{21}$" and "0", and consequently get a misleading favourable assessment.

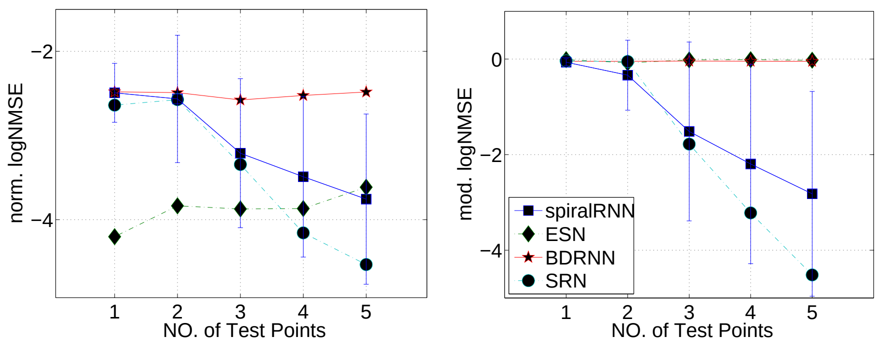
In order to avoid such misleading evaluation, it is advisable to measure the error only at time steps where the corresponding targets are spikes. Assuming a set Λ includes all the time steps t where the data $\hat{x}_t$ are spikes, such that $\Lambda = \{t | \hat{x}_t \simeq 1,\ t \in [1, \tilde{\iota}]\}$ where $\tilde{\iota}$ is the length of the autonomous test, evaluation in *eq-(4.4)* is modified for the *Spike21* task, such that:

$$\varepsilon = \frac{1}{n_\Lambda} \sum_{t \in \Lambda} \log_{10} \left((x_t - \hat{x}_t)^2 / \sigma^2 \right) \tag{4.5}$$

where n_Λ is the number of entries in the set Λ.

The comparison of performance according to the modified *logNMSE* in *eq-(4.5)* is shown in *fig-4.3(b)* and their value as well as standard deviation are listed in *table-4.2*. It is shown in these plots and tables, that the *ESN* has failed to provide an acceptable result in terms of modified *logNMSE* because it can only provide a constant result rather than a spike sequence. Without consideration of those failed simulations, *SRN* has outperformed the other models. As this instability problem of *SRN* has emerged with progressing training step, one can observe in *table-4.2* that the number of failing simulations increases from one test point to the next. The *SpiralRNN* model, on the other hand, has demonstrated its stability in performance and learning speed.

Results comparing these RNN models with complexity $\mathscr{C}_s \simeq 100$ can be found in *fig-4.4* and in *table-4.3*. The *SpiralRNN* model has again manifested its efficiency and stability in terms of the modified *logNMSE*.



(a) *standard* logNMSE (b) *modified* logNMSE

Figure 4.3: *Comparisons in terms of* logNMSE *with Spike21 data, where the model complexity is set to* $\mathscr{C}_s \simeq 50$. *In both diagrams, the X-axis stands for the five test-points, and the Y-axis refers to the* logNMSE *values. Note that neither diagram reveals the unsuccessful simulations of the* SRN *model; both diagrams share the same legend. (a) Comparison in terms of standard* logNMSE*; (b) comparison in terms of modified* logNMSE.

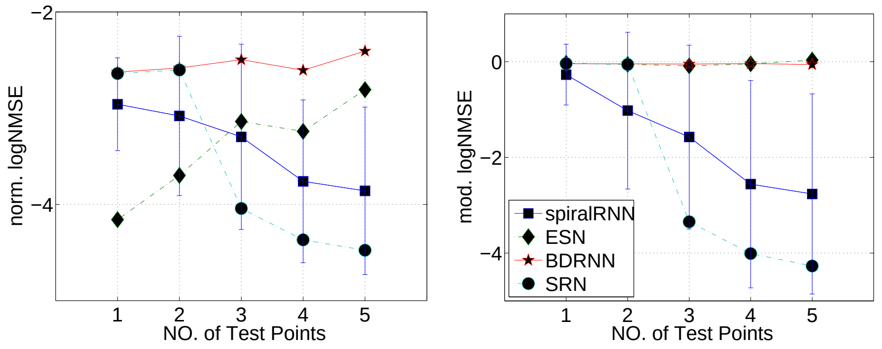
Test Point	*SpiralRNN*		*ESN*		*BDRNN*		*SRN*	
	mean	std.	mean	std.	mean	std.	mean	std.
1	-0.067	0.073	-0.011	0.062	-0.047	0.009	-0.041	0.012
2	-0.337	0.733	-0.080	0.324	-0.046	0.009	-0.050	0.022
3	-1.516	1.872	-0.017	0.055	-0.042	0.009	-1.780	2.015
4	-2.196	2.088	-0.013	0.071	-0.045	0.017	(-3.219)	(1)
5	-2.821	2.142	-0.024	0.049	-0.047	0.009	(-4.518)	(8)

Table 4.2: *Comparison in terms of the modified* logNMSE *and the standard deviation with the Spike21 date set, where the model complexity is* $\mathscr{C}_s \simeq 50$. *Note that, in the experiment with the* SRN *model, an instability is caused by test point No. 4; the value in parentheses in column "std." is the count of invalid simulations.*

4.2.2 *Mackey-Glass* Dataset

The *Mackey-Glass* data set, generated according to *eq-(4.1)*, is a one-dimensional chaotic time series. Parameter τ controls the time delay in the model, which makes the time series combine periodical and chaotic features. See *fig-4.1(c)*.

Test points in experiments with *Mackey-Glass* data are placed at time steps $t = \{10^{[4:0.25:5]}\}$. The length of autonomous tests was set to $\tilde{\iota} = 200$ because the characteristics of chaotic data make the long-term prediction deviate far away from the trajectory, thus removing all meaning from any prediction with 2000 steps.



(a) *standard* logNMSE (b) *modified* logNMSE

Figure 4.4: *Comparisons in terms of* logNMSE *with Spike21 data, where the model complexity is set to $\mathscr{C}_s \simeq 100$. The X-axis stands for the five test points, and the Y-axis refers to the* logNMSE *values. Note that both diagrams have ignored the unsuccessful simulations of the* SRN *model. (a) Comparison in terms of standard* logNMSE*; (b) comparison in terms of modified* logNMSE.

Test Point	*SpiralRNN* mean	*SpiralRNN* std.	*ESN* mean	*ESN* std.	*BDRNN* mean	*BDRNN* std.	*SRN* mean	*SRN* std.
1	-0.271	0.635	-0.032	0.165	-0.044	0.015	-0.039	0.009
2	-1.023	1.639	-0.056	0.172	-0.044	0.013	-0.057	0.043
3	-1.575	1.920	-0.088	0.341	-0.050	0.031	(-3.345)	(1)
4	-2.560	2.167	-0.045	0.094	-0.039	0.013	(-4.010)	(2)
5	-2.765	2.091	-0.033	0.399	-0.060	0.060	(-4.269)	(4)

Table 4.3: *The comparison in terms of the modified* logNMSE *and standard deviation with the Spike21 date set, where all RNN models have the same model complexity value $\mathscr{C}_s \simeq 100$. Note that, in the experiment with the* SRN *model, an instability is caused by test point No. 3; the value in parentheses in column "std." is the count of invalid simulations.*

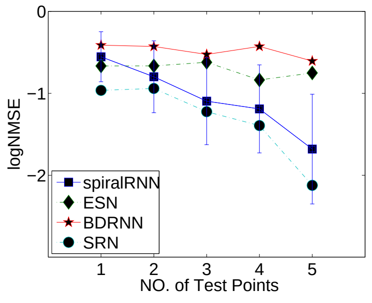
Results of RNN models predicting *Mackey-Glass* are given in *fig-4.5*. It is shown in *fig-4.5* and *table-4.4* that the *ESN* model can only be comparable at the early stages, whereas it fails in long-term prediction because of the fixed structure of the hidden layer. An increase in complexity will help to improve the performance of a *ESN* in general, but it has no improving effect in long-term prediction. The *SRN* model is superior in this task, particularly when the network complexity $\mathscr{C}_s \simeq 50$ in *fig-4.5(a)*; however, when the complexity $\mathscr{C}_s$ increases to 100, it not only loses this advantage over the *SpiralRNN* but also produces worse result than itself at previous time steps, as shown in *fig-4.5(b)*. This reveals in a way the instability of the *SRN* model itself.

SpiralRNN has outperformed the other RNN models in most cases, and this advantage has

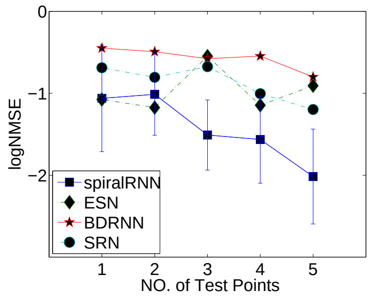
Test Point	*SpiralRNN*		*ESN*		*BDRNN*		*SRN*	
	mean	std.	mean	std.	mean	std.	mean	std.
1	-0.554	0.305	-0.667	0.268	-0.413	0.082	-0.964	0.391
2	-0.799	0.439	-0.666	0.351	-0.429	0.116	-0.941	0.415
3	-1.096	0.532	-0.622	0.277	-0.526	0.110	-1.225	0.513
4	-1.191	0.537	-0.838	0.459	-0.428	0.100	-1.394	0.445
5	-1.681	0.670	-0.752	0.286	-0.606	0.315	-2.123	0.512

Table 4.4: *Comparison in terms of* logNMSE *and standard deviation on data set Mackey-Glass. All models have the model complexity value* $\mathscr{C}_s \simeq 50$*;*

shown to increase with longer training. *ESN* can produce comparable autonomous output in the short term, *i.e.* with small $\tilde{\iota}$ value, but it fails to catch up with the dynamics in the longer term, as it is shown in *fig-4.5(b)*. On the other hand, *SpiralRNN* has retained its advantage over the *ESN* and other models by enlarging the network size (refer to *table-4.5*).



(a) $\mathscr{C}_s \simeq 50$



(b) $\mathscr{C}_s \simeq 100$

Figure 4.5: *Comparison of RNN models in terms of* logNMSE *with different complexity on Mackey-Glass data set. (a) Model complexity* $\mathscr{C}_s \simeq 50$*; (b) model complexity* $\mathscr{C}_s \simeq 100$.

Test Point	*SpiralRNN*		*ESN*		*BDRNN*		*SRN*	
	mean	std.	mean	std.	mean	std.	mean	std.
1	-1.062	0.651	-1.076	0.433	-0.449	0.092	-0.688	0.3580
2	-1.013	0.500	-1.175	0.417	-0.492	0.172	-0.805	0.3520
3	-1.510	0.428	-0.547	0.233	-0.578	0.167	-0.675	0.3150
4	-1.564	0.532	-1.146	0.433	-0.546	0.188	-1.005	0.6480
5	-2.016	0.579	-0.908	0.343	-0.801	0.391	-1.198	0.6380

Table 4.5: *Comparison in terms of* logNMSE *and standard deviation on Mackey-Glass data set. All models have the model complexity value* $\mathscr{C}_s \simeq 100$*;*

4.2.3 *Lorenz* Dataset

Test points in the *Lorenz* data set are time steps $t = \{10^{[3:0.25:5]}\}$, where each autonomous test runs 200 time steps. Comparison results of RNN models with complexity $\mathscr{C}_s \simeq 100$ are illustrated in *fig-4.6* and *table-4.6*. *Fig-4.6* contains three diagrams, each of which represents comparisons with different values of $\tilde{\iota}$ in *eq-(4.4)*.[12] Whilst *table-4.6* has only shown the result when $\tilde{\iota} = 200$, *table-4.7* and *fig-4.7* report the results of comparing RNN models with complexity $\mathscr{C}_s \simeq 200$.

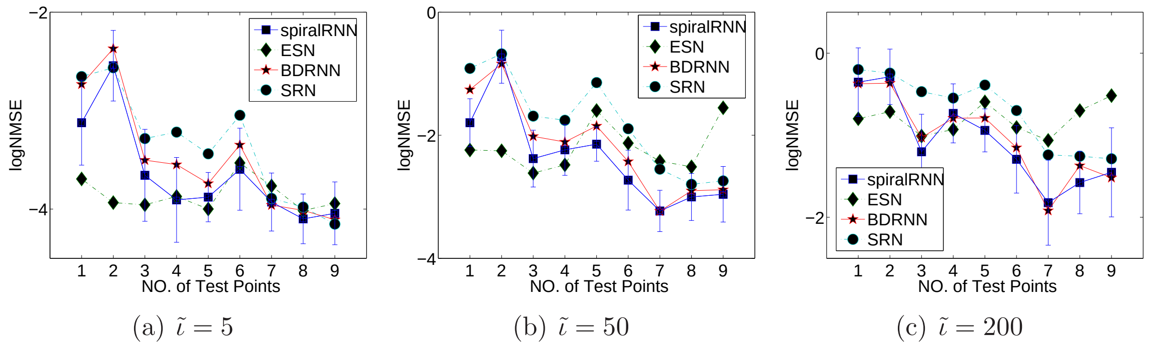


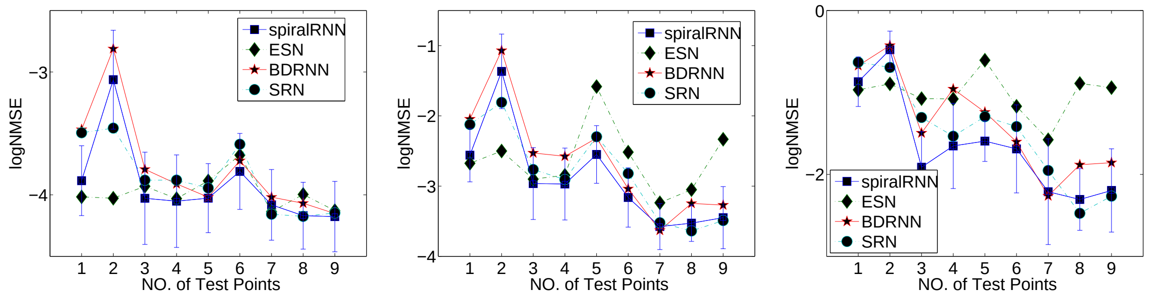
Figure 4.6: *Comparison in terms of* logNMSE *on* Lorenz *data set with various autonomous output lengths* $\tilde{\iota}$. *All models have the model complexity value* $\mathscr{C}_s \simeq 100$.

Test Point	*SpiralRNN*		*ESN*		*BDRNN*		*SRN*	
	mean	std.	mean	std.	mean	std.	mean	std.
1	-0.353	0.417	-0.799	0.301	-0.373	0.388	-0.199	0.467
2	-0.287	0.337	-0.713	0.248	-0.365	0.243	-0.243	0.291
3	-1.202	0.458	-1.009	0.376	-1.028	0.398	-0.470	0.441
4	-0.734	0.359	-0.930	0.424	-0.792	0.377	-0.546	0.346
5	-0.941	0.262	-0.594	0.419	-0.790	0.344	-0.388	0.528
6	-1.294	0.411	-0.906	0.470	-1.150	0.370	-0.699	0.468
7	-1.822	0.518	-1.063	0.448	-1.919	0.505	-1.239	0.644
8	-1.577	0.380	-0.700	0.448	-1.370	0.326	-1.256	0.474
9	-1.452	0.543	-0.517	0.344	-1.518	0.615	-1.287	0.568

Table 4.6: *Comparison in terms of* logNMSE *and standard deviation on Lorenz data set. The model complexity is* $\mathscr{C}_s \simeq 100$, *and the autonomous test length is* $\tilde{\iota} = 200$.

[12] Changing the evaluation variable $\tilde{\iota}$ doesn't affect the simulation result itself. Since autonomous tests in *Lorenz* simulation last 200 time steps, the evaluation with $\tilde{\iota}$ only considers the first $\tilde{\iota}$ items of output data out of the whole autonomous output.

As shown in *fig-4.6(a)*, all RNN models are similarly successful in short-term 5-step prediction. In particular the *ESN* model has already underlined its ability at test point No.1 even though the other RNN models have caught up with the *ESN* at test point No.3. As the value of $\tilde{\iota}$ increases, *i.e.* when longer-term prediction is taken into consideration, the *ESN* cannot maintain the advantage it had in short-term prediction, as shown in *fig-4.6(b)* and *fig-4.6(c)*. The other RNN models with trainable hidden layers have been proven to be superior, especially the *SRN* model and the *SpiralRNN* model.



(a) $\tilde{\iota} = 5$ (b) $\tilde{\iota} = 50$ (c) $\tilde{\iota} = 200$

Figure 4.7: *Comparison in terms of* logNMSE *on* Lorenz *data set with various autonomous output lengths* $\tilde{\iota}$. *All networks have the model complexity* $\mathscr{C}_s \simeq 200$.

Test Point	*SpiralRNN*		*ESN*		*BDRNN*		*SRN*	
	mean	std.	mean	std.	mean	std.	mean	std.
1	-0.869	0.302	-0.968	0.382	-0.673	0.311	-0.6310	0.329
2	-0.478	0.227	-0.895	0.480	-0.426	0.235	-0.6950	0.306
3	-1.912	0.434	-1.076	0.992	-1.495	0.407	-1.3050	0.337
4	-1.653	0.520	-1.079	0.572	-0.957	0.350	-1.5340	0.609
5	-1.594	0.248	-0.607	0.475	-1.240	0.270	-1.2950	0.277
6	-1.689	0.537	-1.169	0.569	-1.603	0.455	-1.4170	0.506
7	-2.211	0.645	-1.577	0.527	-2.263	0.594	-1.9520	0.635
8	-2.306	0.376	-0.890	0.413	-1.882	0.385	-2.4750	0.578
9	-2.196	0.508	-0.941	0.341	-1.857	0.510	-2.2650	0.455

Table 4.7: *Comparison in terms of* logNMSE *and standard deviation on Lorenz data set. All models have the model complexity* $\mathscr{C}_s \simeq 200$, *and the autonomous test length reads* $\tilde{\iota} = 200$.

4.2.4 Discussion

The novel recurrent neural network model *SpiralRNN* has shown its efficiency and stability in the above simulations. In most cases, it provides the best performance among the state-of-art architectures. An example of autonomous output of the *SpiralRNN* model on the chaotic *MackeyGlass* time series is shown in *fig-4.8* where the difference between autonomous output and target data can hardly be identified till around time step 400.

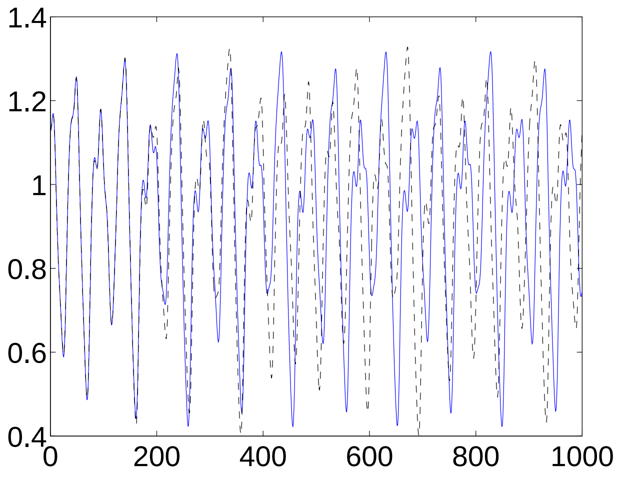


Figure 4.8: *The autonomous output of a trained* SpiralRNN *model vs. corresponding target data. The solid line is autonomous output, and the dashed line is target data.*

In simulations, shortcomings of other conventional RNN models have revealed. The *ESN* model is simple to implement, and in much literature it has been proven to be powerful in off-line learning in many literatures. However, it doesn't provide good results in on-line training, an important requirement in distributed sensor network applications. Furthermore, as indicated by Jaeger, the *ESN* model requires further parameter tunning to control the maximum absolute eigenvalue of the hidden-weight matrix, which is contrary to the requirement $< \Re 1 >$ for easy deployment of sensors, mentioned on *page-3*. The *SRN* model has general recurrent layer structure and theoretically the most suitable structure for dynamics modeling, but it suffers from instability as shown in the simulations with *Spike21* data. The *BDRNN* has only shown comparable performance in some simulations with the *Lorenz* data set.

4.3 *MouseTracking* with *SpiralRNN*s

This section present a piece of software which applies the *SpiralRNN* model to a real-world prediction application. The software is called "*MouseTracking*" as it tries to trace the movements of a computer mouse controlled by a human user. This toy software has been coded and implemented in the MatLab environment, and its MatLab scripts can be found in *appendix-C*. *Fig-4.9* shows the *MouseTracking* interface, with the upper subplot depicting the growing trajectory (in blue) as well as the autonomous prediction (in red) and the lower subplot showing the whole history up to now on a logarithmic scale. *MouseTracking* accepts keyboard commands as listed in *appendix-C*.

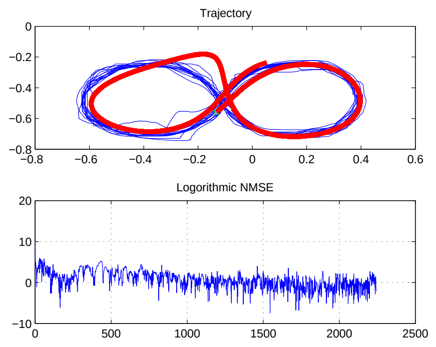


Figure 4.9: *MouseTracking's graphical interface.*

4.3.1 The *MouseTracking*

When the software is running, the human user continuously move the computer mouse such that cursor on the computer monitor runs along a given closed trajectory. The movement is noisy and subject to leaps due to the sensitivity of the optical mouse to the underlying surface. The computer, acting as a sensor, traces and samples the mouse movements on the computer screen at a fixed rate. After some pre-processing, the position data are sent to the recurrent neural network (RNN) model in the computer so that the RNN model can train the network parameters and build an approximated model. Training takes place online, *i.e.* the parameters are adjusted as soon as new measurement data for each time step are available. Test points for this *MouseTracking* software are $\{[1:1:5]*10^3\}$.[13] When time step t coincides with a test point, meaning also that the RNN model has now been trained with t data items altogether, the autonomous test will start and generate an output sequence of coordinates. The autonomous test will generate 1000 steps, and its output is expected to coincide with the user's trajectory. The *pseudo*-code for *MouseTracking* is given in *table-4.8*.

4.3.2 Training Data

As mentioned, the optical mouse is controlled by a human user so that the cursor traverses along the figure-of-8 trajectory shown in *fig-4.10(a)*. Every 1/100 second[14], the position of

[13] These test points are {1e3, 2e3, 3e3, 4e3, 5e3}. As it will be mentioned in the later text, period of trajectory employed in the simulation is roughly around 130, therefore, at test-point $t = 1e3$, the whole trajectory has presented only 8 times in the training data.

[14] This value is manually set and is empirical.

```
initialize the neural network model and other configuration;
set the vector of test points;
for each time step $t$
    measure the coordinates of the position of the mouse;
    implement the training of network parameters based on the observation data
    if $t$ = one of test points
        set the latest observed data as input of network;
        implement autonomous prediction and generate the autonomous output;
        (autonomous output is stored for evaluation purpose)
    end
end
```

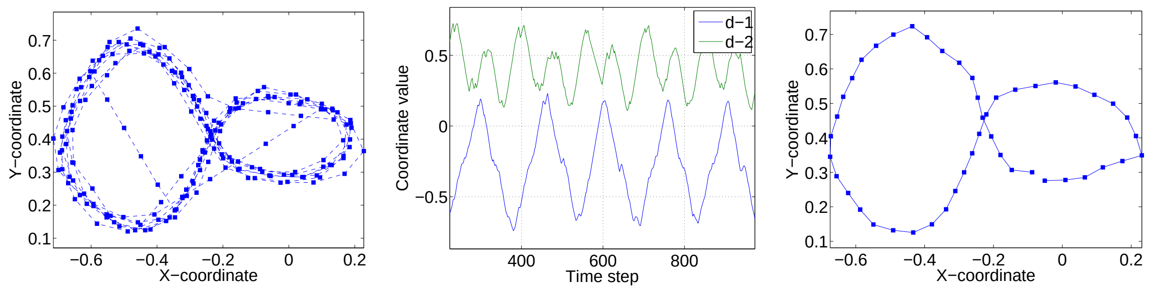
Table 4.8: *The pseudo code for the MouseTracking.*

the mouse cursor is measured by the computer. Such a measurement will not be used directly as the input for the learning model. Instead, some pre-processing of data is required, because the user's varying hand speed makes the raw measurement data less predictable. In pre-processing, the path of the computer mouse is divided into segments of identical length. The boundary points that split the mouse path into segments of identical length[15] are concatenated together after their coordinates have been normalized. The boundary points subdividing the mouse path into identical stretches are taken as time-step points, as if the computer mouse were moving at a constant speed. This is shown in *fig-4.11*, where the hollowed arrow indicates the movement direction, the dashed line is the movement path of cursor, circles represent the data measured by computer every 1/100 second, and black dots refer to boundary points that segment the movement path of cursor. Note that the distance between circles is various, due to the human-controlled moving of mouse, and once the movement path is known the positions of measured point is dispensable.

The normalization of boundary points' coordinates is implemented such that coordinates on the computer monitor screen cover the range $[-1, 1]$ on both axes and the center of the screen lies at the origin. The identical length mentioned above is set to "0.02" after normalization. Example values of two coordinates of the trajectory are given in *fig-4.10(b)*, where the trajectory period can be observed, being roughly equal to 130 time steps. Dots in the figures represent some of the boundary points[16], and dashed curves illustrate the trajectories. *Fig-4.10(a)* exhibits a couple of leaps in the trajectory due to a malfunction of the optical mouse.

[15] The path length between two boundary points is equal to the predefined identical length, even if the distance between boundary points occasionally differs from the length due to the curvature and noisiness of the trajectory.

[16] For clarity's sake, only every fifth points is shown in the diagram.



(a) "8" shape trajectory (b) coordinates of trajectory (c) prototype trajectory

Figure 4.10: *Example of a trajectory and its coordinate values and prototype pattern: (a) Trajectory of "8" shape; (b) coordinate values of trajectory;. (c) prototype trajectory (It is useful in evaluation).*

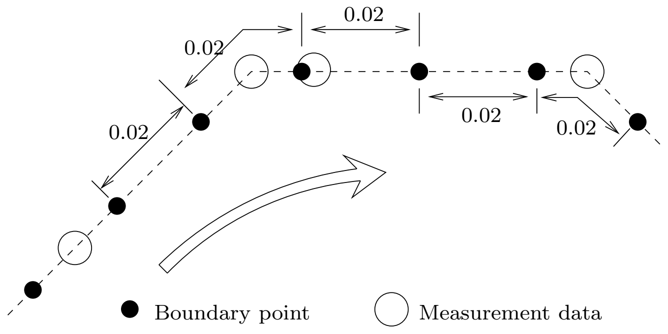


Figure 4.11: *Segmentation by boundary points in MouseTracking. The hollowed arrow indicates the movement direction, the dashed line is the movement path of the cursor, circles represent the position measured by computer in every 1/100 second, and black dots refer to boundary points that segment the movement path of cursor. The identical length between boundary points is "0.02". The positions of these boundary points are taken as the training data at each "time" step.*

4.3.3 Competing Models

In this simulation, *SpiralRNN* is compared to *ESN* and *BDRNN* (*SRN* model is omitted because of the instability). Configuration and initial parameter setting follow the description in *section-4.1*. Note that the number of input or output neurons in the network equals the dimension of the data (2 in this case).

4.3.4 Evaluation Methods

The *MouseTracking* performance evaluation differs from other tests. Recall that the mouse trajectory, as controlled by a human user, contains leaps as shown in *fig-4.10(a)*. The target for comparison is a prototype trajectory extracted from the original data. The prototype trajectory is depicted in *fig-4.10(c)* with length of 47 and denoted as $\mathscr{P} = \{p_1, p_2, \dots\}$, where only every third point is shown for clarity's sake. For each prediction point x, the prediction error is defined as the square of the shortest distance from the prediction point to the prototype trajectory, as shown in *eq-(4.6)*. The evaluation error ε for the autonomous output (of length 1000) at a particular test point is given by *eq-(4.7)* which takes the mean value over the prediction length. For statistics reasons, the final evaluation error will be the mean over 30 simulations.

$$\epsilon_i = \min_k \left\{ \sum_d (x_i(d) - p_k(d))^2 \right\} \tag{4.6}$$

$$\varepsilon = \overline{\epsilon_i}, \quad i = 1, \dots, 10^3 \tag{4.7}$$

Note that the above evaluation method is invalidated if the predicted trajectory stays in the neighborhood of fixed points close to the prototype trajectory. In order to evaluate the result in another light, the frequency analysis results based on Fourier transformation are also given. Given the autonomous output trajectory at each test point, one can use the discrete Fourier transform (DFT) for estimating the frequency of the output time series; MatLab provides the discrete Fourier transformation through the **fft** command. As the length of the autonomous trajectory is 1000, the 1024-point DFT is calculated as in *eq-(4.8)*:

$$Y = \frac{1}{1000}\mathbf{fft}(X, 1024) \tag{4.8}$$

where X is the time series in question and Y is the output vector of the discrete Fourier transform of length 1024. For clarity's reason, only the frequency spectrum range from 0 to 0.03 is shown below since the dominating frequencies of the original trajectory are located in this range; for example the dominant frequency of the trajectory is $f^* \simeq 1/130 \simeq 0.0077$. The final result of the frequency analysis will again be averaged over 30 simulations. To properly show the frequency analysis result in figure, the MatLab code[17] is given in *table-4.9*. For the sake of clearness, only the frequency spectrum within the range $[0, 0.03]$ is displayed.

4.3.5 Results & Discussion

Table-4.10 reports the evaluation error of neural network models at all test points, where the model complexity of neural network is set to $\mathscr{C}_s \simeq 100$. The corresponding comparison

[17] The MatLab code derives from the sample code given in the function reference of command **fft** under MatLab environment.

```
Fs = 1;  % (sample frequency)
nn = 1024;
fr = Fs/2*linspace(0, 1, nn/2);  % (frequency axis ticks)
plot(fr,  2*abs(Y(1:n/2)));
```

Table 4.9: *The MatLab code for plotting the frequency figure.*

with network complexity $\mathscr{C}_s \simeq 200$ is given in *table-4.11*. Both tables report similar results, namely that *SpiralRNN* has an advantage over *ESN* and *BDRNN*, in particular after long-term training of the model.

Test Point / Model	No.1	No.2	No.3	No.4	No.5
SpiralRNN	-1.49	-1.99	-2.42	-2.44	-2.51
ESN	0.15	-1.07	-1.84	-1.86	-1.75
BDRNN	-1.69	-2.01	-1.99	-2.09	-2.20

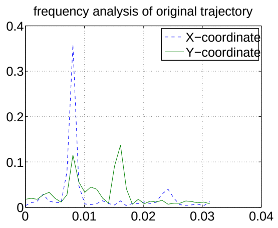
Table 4.10: *Comparison of evaluation error ε at each test point. The neural network models have complexity $\mathscr{C}_s \simeq 100$. (The smaller the error value is, the better the prediction result is.)*

Test Point / Model	No.1	No.2	No.3	No.4	No.5
SpiralRNN	-1.19	-2.01	-2.51	-2.46	-2.78
ESN	-1.88	-1.28	-1.67	-2.05	-1.89
BDRNN	-1.49	-2.07	-2.12	-2.15	-2.20

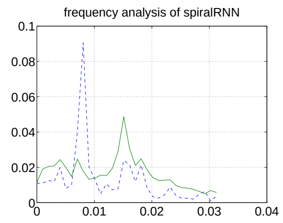
Table 4.11: *Comparison of evaluation error ε at each test point. The neural network models have complexity $\mathscr{C}_s \simeq 200$. (The smaller the error value is, the better the prediction result is.)*

The frequency analysis of a simulation with model complexity $\mathscr{C}_s \simeq 100$ is provided in *fig-4.12*, where the frequency analyses of X-coordinate time series are depicted as blue dash curves and the ones for Y-coordinates value are given as green solid curves. *SpiralRNN* produces (ref. *fig-4.12(b)*) autonomous output trajectories with the frequency spectrum similar to the frequency spectrum of the target data (ref. *fig-4.12(a)*), while the other models cannot. The frequency analysis for models with larger network size $\mathscr{C}_s \simeq 200$ has confirmed the superior of the *SpiralRNN* model, which is given in *fig-4.13*.

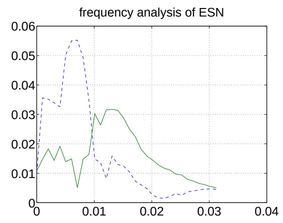
Fig-4.14 respectively depicts examples of autonomous output trajectories at test points No.$3, 4, 5$ using the *SpiralRNN* model with complexity $\mathscr{C}_s \simeq 100$. These output examples were extracted from the same simulation. Note that, in *fig-4.14(b)* of the No.4 autonomous



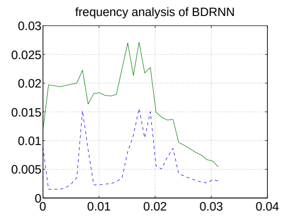
(a) Target



(b) *SpiralRNN*

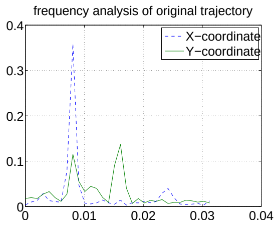


(c) *ESN*

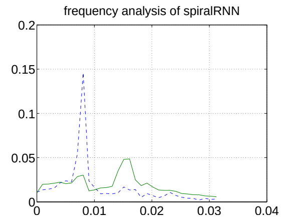


(d) *BDRNN*

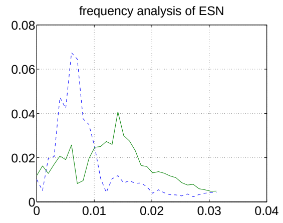
Figure 4.12: *Frequency analysis of the output trajectory at test point No.5 of competing neural network models with complexity $\mathscr{C}_s \simeq 100$. Note that the vector value at frequency "zero" is trivial and is therefore omitted in diagrams, and that the period length of data is 130 therefore the dominant frequency is $\frac{1}{130} \simeq 0.008$.*



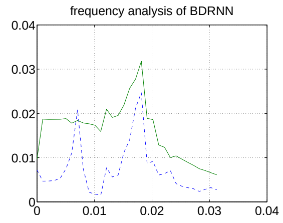
(a) Target



(b) *SpiralRNN*



(c) *ESN*

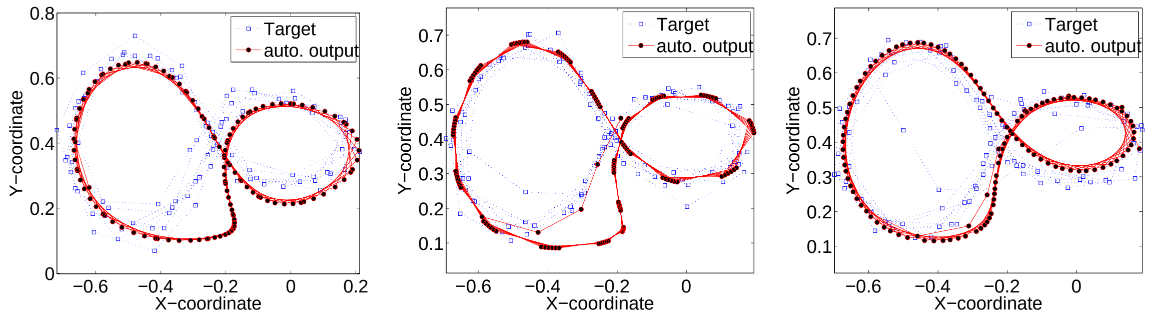


(d) *BDRNN*

Figure 4.13: *Frequency analysis of the output trajectory at test point No.5 of competing neural network models with complexity $\mathscr{C}_s \simeq 200$. Note that the vector value at frequency "zero" is trivial and is therefore omitted in diagrams, and that the period length of data is 130 therefore the dominant frequency is $\frac{1}{130} \simeq 0.008$.*

test, the first loop of the autonomous output coincides with that of the target, even though it deviates later to a slightly deformed limit-cycle. *Fig-4.14(c)* of the No.5 autonomous test depicts a rather close prediction of the trajectory.

Similar performance and result have been observed for the *MouseTracking* software with other shapes of trajectory, *e.g.* triangle and square. *SpiralRNN* has demonstrated to be efficient and stable in learning data from the physical world.



(a) No.3 (b) No.4 (c) No.5

Figure 4.14: *Typical examples of autonomous output trajectories produced by the* SpiralRNN *model with complexity* $\mathscr{C}_s \simeq 100$. *The red solid curves are the autonomous trajectories predicted by the* SpiralRNN *and the blue dashed curves refer to the original trajectories during the previous* 1000 *time steps before the respective test point. (a) Autonomous output at the No.3 test point; (b) autonomous output at the No.4 test point; (c) autonomous output at the No.5 test point.*

4.4 Conditional Prediction with *SpiralRNN*s

There are many applications where a system evolves differently depending on one or more external triggering "context" signals. A typical example in discrete manufacturing is a robot manipulating bottles transported by automated guided vehicles (AGV). Depending on sensor signals indicating (i) the presence of a bottle, (ii) whether the bottle is already filled or empty and (iii) which liquid to fill in, the robot starts different sequences of actions like filling liquid from different sources or ignoring already filled bottles. Installation and reconfiguration of automation system will be greatly simplified by those controllers which can be easily trained to perform these sequences of actions depending on sensor information.

In the context of this section, the term "conditional prediction" refers to those applications where the learning model is trying to approximate the dynamics model with the assistance of an external triggering signal. The information provided by such additional signals can improve the efficiency and accuracy of the control, which is useful in at least three categories:

1. Comparison of a predicted trajectory with an actual trajectory for diagnostic purposes;
2. On-line training of a specific sequence of actions for control purposes;
3. Substitution of missing trajectory data by predicted ones.

SpiralRNN contributes to conditional prediction problem by providing an efficient and stable learning model. A warehouse application scenario will be outlined below and the implementation of the *SpiralRNN* model in conditional prediction [86] will be described. Simulation and result as well as analysis are given afterwards.

4.4.1 Scenario

Consider an AGV system in a warehouse on 11×11 grid with various commodities in it, as shown in *fig-4.15*. Inside, each type of commodity has its own unique cabin, associating a unique path[18] which connects the entrance to this particular cabin. A trailer or trolley runs through the grid, and ships a product to its assigned cabin whenever the product arrives at the entrance. Differences between products can be digitalized as each production type is associated with a unique value, and this additional information is available as soon as the product shows up at the entrance.

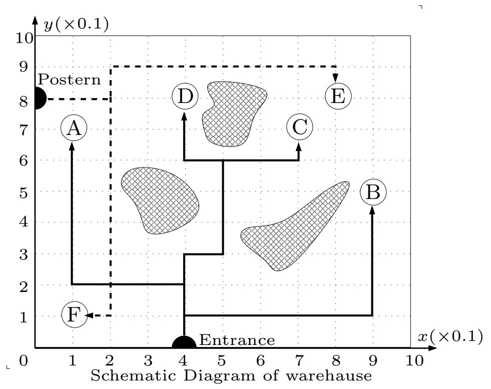


Figure 4.15: *The schematic diagram of a warehouse in* 11×11 *grid. Product cabins are denoted by capital letters with circle, obstacles are shaded areas. For example, the cabin for product A is located at* $(0.1, 0.7)$*, product B at* $(0.9, 0.5)$*, product E at* $(0.8, 0.8)$*. Products can enter the warehouse via the entrance* $(0.4, 0)$ *or the postern* $(0, 0.8)$*. Trajectories of products coming into the warehouse via the entrance are shown in solid lines, and trajectories via postern are shown in dashed lines.*

Imagine each trolley has an "intelligent sensor" attached. When the trolley is assigned a job to convey product A to its respective cabin, the sensor is given the associated corresponding additional information. This additional information identifies the production type, and can be digitalized as ϕ_A. During transport, the sensor measures the corresponding trajectory $\vec{p}^{(A)} = \left\{p_1^{(A)}, \ldots, p_n^{(A)}\right\}$ of product A. The trajectory data will be used to train the embedded model of the sensor node, such that it satisfies $p_{t+1}^{(A)} = f^*(p_t^{(A)}, \phi_A)$, with f^* symbolizing the model to be trained and $p_t^{(A)}$ referring to the corresponding entry

[18] As shown in *fig-4.15*, these are bifurcations in these unique paths.

of trajectory $\vec{p}^{(A)}$ at time step t. Training stops when the product has been transported to the assigned cabin, which is indicated by sending a terminating signal to the sensor. Assigned a new job (for example, to convey a product D to its cabin), the trolley is given a new trigger value ϕ_D, and has to go through another path $\vec{p}^{(D)} = \left\{p_1^{(D)}, \ldots, p_n^{(D)}\right\}$. In this case, the embedded model has to be switched to another attractor such that it satisfies $p_{t+1}^{(D)} = f^*(p_t^{(D)}, \phi_D)$. Therefore, during the transport of different products, the embedded model of the sensor is trained to modulate different attractors of different trajectories, as shown in *eq-(4.9)*. After a certain time of learning, the embedded model is supposed to be able to tell the trolley the proposed entire trajectory for a particular product, as soon as the ϕ value is known.

$$p_{t+1}^{(i)} = f^*\left(p_t^{(i)}, \phi_i\right), \quad i = A, B, \cdots \tag{4.9}$$

4.4.2 Conditional Prediction with RNNs

Recalling the general model of a recurrent neural network, as shown in *eq-(4.10)* and *eq-(4.11)*,

$$\mathbf{s}_t = \mathscr{H}(W_t, \mathbf{s}_{t-1}, \mathbf{x}_{t-1}) \tag{4.10}$$
$$\mathbf{x}_t = \mathscr{G}(W_t, \mathbf{s}_t) \tag{4.11}$$

one can further simplify them by omitting the intermediate hidden-state vector $\mathbf{s}_t$, as shown in *eq-(4.12)*.

$$x_t = f\left(x_{t-1}\right) \tag{4.12}$$

In order to apply recurrent neural network models in conditional prediction, the following adjustments are required:

- As *eq-(4.9)* is not analogous to the general model of RNN in *eq-(4.12)*, a rearrangement is required to shape the *eq-(4.9)* into *eq-(4.13)* for reasons of modeling. Now, a RNN model can be applied to the conditional prediction by taking x_t as the tuple combining the trajectory data p_t and the corresponding ϕ value, *i.e.* $x_t = [p_t, \phi]$.

$$\left[p_{t+1}^{(i)}, \phi_i\right] = f^*\left(p_t^{(i)}, \phi_i\right), \quad i = A, B, \cdots \tag{4.13}$$

- The initialization condition of a trajectory consists in the value setting of starting point p_0 and the trigger signal ϕ. While the ϕ value is directly determined by the product type, the starting point value p_0 can be manipulated. This starting point is virtual and doesn't exist in the actual trajectory. To achieve stable reproductions of the associated trajectory patterns, this virtual starting point p_0 for the particular trajectory pattern should remain unchanged. There are two options for the initialization of p_0, where in both cases the p_0 value of particular trajectory pattern has to remain unchanged after initialization, as if it was an actual part of the corresponding trajectory pattern. They are:

Scheme-1 The p_0 value is set to zero for all trajectories without exception.

Scheme-2 The p_0 value is randomly initialized when the particular product occurs for the first time.

Note that the introduction of the virtual staring point p_0 magnifies the impact of the initial tuple x_0 on the learning. Similar effect can be achieved by setting the hidden-state vector $\mathbf{s}$ with a particular randomly-predefined value whenever model starts the training of a particular trajectory pattern. In this thesis, the former solution (with the introduction of p_0) is taken.

With the new model in *eq-(4.13)* and the completion of training data (by inserting the virtual starting point), recurrent neural network (RNN) models can be applied in conditional prediction problems. The implementation comprises the initialization step, the training step and the testing step.

Initialization:

Whenever a product appears at the entrance (or the postern), no matter whether it is for training or testing, the sensor triggers the signal ϕ according to product type and starts the initialization. In the initialization, the hidden-state vector of the neural network will be reset to zero in order to avoid any influence from the past. The tuple $x_0 = [p_0, \phi]$ is only initialized when the product type is presented at the first time. In this way, the initial tuple x_0 is unique and constant for the corresponding trajectory. The *pseudo*-code for the initialization is given in *table-4.12*.

Training:

At time step $t+1$, for instance, the target tuple is constructed with the latest data: $\hat{x}_{t+1} = [p_{t+1}, \phi]$. The neural network iterates itself according to *eq-(4.10)* and *eq-(4.11)* with the previous tuple $\hat{x}_t$ as input. Note that, by teacher forcing, x_{t-1} (network output) is replaced by $\hat{x}_{t-1}$ (data) in *eq-(4.10)*. As in an on-line learning mode, network parameters are updated at each time step, and the output x_{t+1} is compared to the new-constructed target tuple $\hat{x}_{t+1}$. The on-line training is again based on an extended Kalman filter (EKF) combined with the real-time recurrent learning (RTRL). When the trolley arrives the target cabin, the attached sensor triggers a termination signal, and stops the training process. The *pseudo*-code for training with any trajectory pattern is given in *table-4.13*.

Testing:

In testing, the product type is known, as is the ϕ value. The task is to reproduce the corresponding trajectory of a particular product for monitoring purposes when the corresponding ϕ value is given. Again, the initialization is carried out before everything else, setting up the initial tuple x_0 and eliminating from the hidden state any remaining information.

```
*obtain* the corresponding $\phi$ value of a trajectory pattern;
*reset* the value of hidden-state vector $\mathbf{s}$ to zero;
**switch** initialization scheme of $p_0$
**case 1:** (scheme-1)
    *set* value of $p_0$ to zero
**case 2:** (scheme-2)
    **if** the trajectory pattern occurs for the first time
    **then**
        randomly *initialize* the $p_0$ value over the range of (-1,1);
        *save* this initial value for this pattern;
    **else**
        *restore* the $p_0$ value from the corresponding saved value;
    **end**
**end**
*construct* the initial tuple $x_0 = [p_0, \phi]$
```

Table 4.12: *The pseudo code for the initialization in conditional prediction*

```
*implement* the initialization step and *obtain* the initial tuple $x_0$

**for** each data $p_{t+1}$ in the trajectory pattern $\vec{p}$
    *construct* a new tuple $\hat{x}_{t+1} = [p_{t+1}, \phi]$
    *set* the previous tuple $\hat{x}_t$ as the network input;
    *iterate* the neural network according to *eq-(4.10)* and *eq-(4.11)*;
    *calculate* the residual $\delta = \hat{x}_{t+1} - x_{t+1}$;
    *adjust* parameters according to the $\delta$ value based on the EKF;
**end**
```

Table 4.13: *The pseudo code for training in conditional prediction.*

With the initial tuple x_0 from the initialization step, the neural network iterates according to *eq-(4.10)* and *eq-(4.11)* as it does in training step. Instead of comparing with the target, as it did in training, the network output in testing phase will be used as network input in the next iteration of this autonomous test. Such autonomous iterations continue until the stopping criterion is satisfied, usually by the length of the autonomous test reaching the predefined value l_p. The *pseudo*-code for testing of one trajectory pattern is given in *table-4.14*.

```
*implement* initialization step and *obtain* the initial tuple $x_0$
**while** the iteration step $i_t$ doesn't exceed value of $l_p$
    *set* the tuple $x_t$ as the network input;
    *iterate* the neural network according to *eq-(4.10)* and *eq-(4.11)* and
    *generate* the network output $x_{t+1}$;
    *increase* the $i_t$ value by one;
**end**
*concatenate* all autonomous outputs to form the predicted trajectory $\vec{x} = \{x_1, \ldots, x_n\}$;
```

Table 4.14: *The pseudo code for testing in conditional prediction*

4.4.3 Experimental Settings

Data

The trajectory data set is directly extracted from the trajectory coordinates of different products in the warehouse scenario shown in *fig-4.15*. It is assumed that the trolley, so as to convey products, moves at a constant speed[19] and that, within a sampling-time interval of the sensor, the trolley moves from one grid point to one of four neighboring grid points, as long as they are not blocked by obstacles. The ϕ values for trajectory patterns A to F are respectively set to $\{1, -1, 0.5, -0.5, 2, -2\}$. The value of p_0 can be set according to either of two initialization schemes, to show the effects of different initializations on performance. The length of each trajectory (excluding the virtual starting point) is set to $l_p = 11$ but may also have different values for different trajectories. The number of trajectory patterns n_p in each simulation varies, ranging from 2 to 6. Choice of trajectory patterns will usually start with product A, *e.g.* trajectories of products A, B and C will be chosen when $n_p = 3$. Data are corrupted by normally distributed noise satisfying $\mathscr{N}(0, 10^{-2})$.

Comparison models

The performance of *SpiralRNN* models will be compared to that of conventional neural network models, such as echo state networks (*ESN*), block diagonal recurrent nets (*BDRNN*) and the classical multi-layer perceptrons (*MLP*) with three layers. Recall that the EKF algorithm dominates the training effort and that the complexity of the EKF algorithm is determined by the number of parameters in the model. Neural network models will be compared to each other under the condition that all models have similar network sizes. As the number of input/output neurons of models is dependent on the dimension of data and is therefore fixed (in this case, input entries include two for the trajectory coordinates and one for the ϕ value), one can change the number of hidden nodes in order to change the model complexity $\mathscr{C}_s$. *Table-4.15* lists the number of hidden neurons in the competing models with different network

[19] When the trolley is not operated at constant speed, dividing the trajectories into segments of identical length can fake constant speed, as is done with the *MouseTracking* data in *section-4.3.2*.

sizes, and the listed settings will be used in simulations.

$\mathscr{C}_s$ \ model	*SpiralRNN*	*ESN*	*BDRNN*	*MLP*
100 weights	12	30	12	14
200 weights	24	63	24	28
300 weights	36	96	38	42

Table 4.15: *Numbers of hidden neurons in neural network models with various network complexities (different $\mathscr{C}_s$ values). Note that the values in the first column refer roughly to the number of trainable weights of the networks.*

Without explicit notification, parameters in all models will be initialized according to the normal distribution $\mathscr{N}(0, 10^{-2})$. For reasons of easy deployment, the construction of the *SpiralRNN* model follows *chapter-3*. The number of hidden units will be the same as the number of input nodes, where all hidden units have an identical number of hidden nodes. The configuration of the *ESN* model follows [50]. Its hidden weight matrix W_{hid} is initialized randomly over range $[-1, 1]$ with the sparsity value 95%; afterwards the matrix is rescaled such that the maximum eigenvalue of W_{hid} is equal to 0.8. The *BDRNN* model will take the scaled orthogonal version (refer to [53]), where entries of each of the 2×2 sub-block matrices are determined by two variables w_1 and w_2 satisfying $w_1^2 + w_2^2 \leq 1$. Similar to the other models, the three-layer *MLP* model has 3 input and output nodes, whereas the number of hidden nodes is adjusted according the network complexity.

Training and testing

Training will be carried out in rounds, and it is called "training round". In each training-round, the model is trained with all trajectory patterns, one by one in random order. After n_t rounds of training, a test round begins. In a test round, the trained model tries to reproduce the seen trajectory patterns, given the corresponding ϕ values and their respective virtual starting points, Training and testing follow the respective description in *section-4.4.2*.

Evaluation

Even though the data set is 3-dimensional, the evaluation focuses only on the trajectory entry set, named Γ, rather than the entry for ϕ. The evaluation error ε at each test round is calculated according to *eq-(4.14)* and *eq-(4.15)*. Note that $\epsilon_{i,t,k}$ refers to the error of l_p-step-ahead prediction when $t = l_p$, and that $\overline{\epsilon_{i,t,k}}$ in *eq-(4.15)* refers to taking the mean value of ϵ over all indices i, t and k. For statistics reasons, the final result takes the average value as well as the standard deviation value over 30 simulations.

$$\epsilon_{i,t,k} = (\hat{x}_{i,t,k} - x_{i,t,k})^2, \quad i \in \Gamma, t \in [1, l_p], k \in [1, n_p] \tag{4.14}$$

$$\varepsilon = \overline{\epsilon_{i,t,k}} \tag{4.15}$$

Because of the regular grid where the interval between grid lines equals 0.1 (refer to *fig-4.15*), a prediction is considered to be matched if the absolute prediction error $|\hat{x} - x|$ is smaller than 0.05. Therefore, the threshold value θ for evaluation is set to logarithmic value of 0.05, as in *eq-(4.16)*. This threshold value will be used as an indication of fulfillment of performance in the following discussion.

$$\theta = \log(0.05^2) \simeq -2.6 \tag{4.16}$$

4.4.4 Results

Simulations have been implemented with two initialization schemes for p_0, as described on *page-66*. In all tasks, simulations are repeated 30 times for statistics reasons. Evaluation is done by considering the mean value and standard deviation value of results over 30 simulations. All numbers in tables are base-10 logarithm values of the respective ε. In the histograms, the X-axis represents the ε value in 10^{-3} units, and the Y-axis shows the occurrence frequency of the respective ε value.

Results with p_0 initialization *scheme-1*

With *scheme-1*, p_0 is set to all zeros without exception. Results in *table-4.16* report the performance of mentioned neural network models in different tasks where the number of network parameters varies from 100 to 300 and the n_p value is set respectively from 3 to 5. Example histograms of ε values are given in *fig-4.16* (Note the different scales of the X-axis and the Y-axis in different sub-figures). Each sub-figure in *fig-4.16* depicts the histogram of the particular neural network model (with settings $\mathscr{C}_s = 200$, $n_t = 30$ and $n_p = 4$). The *SpiralRNN* model outperforms the others in these tests even though the *MLP* model has given relatively comparable and stable (ref. *fig-4.16(c)*) performance, whilst the *BDRNN* suffers from the outlier in the result (refer to *fig-4.16(d)*) and the *ESN* model hasn't be able to deliver a good result.

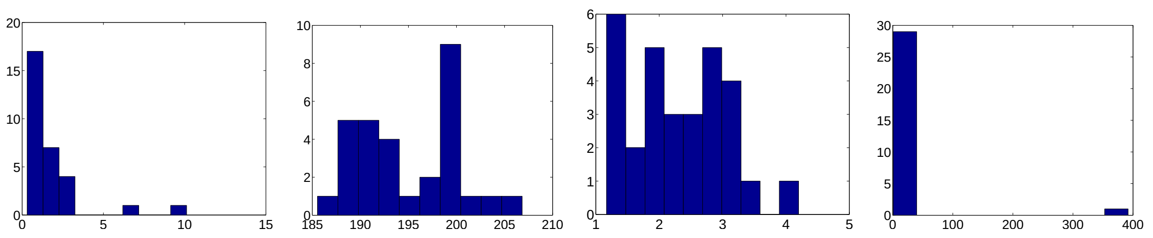
Table-4.17 depicts the results of the *SpiralRNN* model in different tasks (*w.r.t.* network size and n_p value) after n_t training rounds. Each column in the table represents one task with a particular network size and a particular n_p value. A simulation result is not available for all n_t values, and symbol "-" means no simulation was performed. Bold font marks the first result in each column better than threshold value θ (if there is). The performance of *SpiralRNN* improves if it sees more data, as the value of n_t increases. But for complicated tasks (with a bigger value of n_p), the model cannot deliver satisfying results even after $n_t = 110$ training rounds.

Results with p_0 initialization *scheme-2*

Table-4.18 presents the performance results from competing models which have been trained $n_t = 20$ rounds, where neural network models are set to different network sizes from 100 to 300 and are supposed to reproduce different numbers of trajectory

task	100 weights, $n_p = 3$				200 weights, $n_p = 4$				300 weights, $n_p = 5$			
model \ n_t	10	30	50	70	10	30	50	70	10	30	50	70
SpiralRNN	-2.31	-2.75	-2.93	-2.99	-2.15	-2.74	-3.19	-3.31	-2.02	-2.50	-2.63	-2.87
ESN	-0.71	-0.71	-0.72	-0.72	-0.70	-0.70	-0.71	-0.71	-0.69	-0.73	-0.73	-0.73
BDRNN	-2.02	-2.23	-2.29	-2.38	-1.90	-1.71	-2.37	-2.49	-1.82	-2.25	-2.36	-2.24
MLP	-2.24	-2.42	-2.56	-2.56	-2.07	-2.63	-2.69	-2.70	-1.67	-2.12	-2.39	-2.55

Table 4.16: *Comparisons of evaluation error ε defined in eq-(4.15) of different models in different tasks (w.r.t. network sizes and number of trajectory patterns n_p). Note that all values are on a logarithmic scale and initialization scheme-1 is used.*



(a) *SpiralRNN* (b) *ESN* (c) *MLP* (d) *BDRNN*

Figure 4.16: *Histograms of evaluation error ε (over 30 simulations) of neural network models with 200 network parameters with $n_p = 4$ trajectory patterns and $n_t = 30$ training rounds. The X-axis refers to the value of evaluation error in unit of 10^{-3}, and the Y-axis shows the occurrence frequency of respective ε value. Note the different scales of the X-axis and Y-axis in different sub-figures.*

patterns. A comparison of histograms of these models (with settings $\mathscr{C}_s = 200$, $n_t = 20$ and $n_p = 4$) is shown in *fig-4.17*. The *SpiralRNN* model has surpassed all the other mentioned models in terms of accuracy (refer to *table-4.18*) and stability (refer to *fig-4.17*). The *ESN* model has failed in most of the cases, and the *BDRNN* has again seen outliers in simulations. The *MLP* has shown slow convergence, and performance degrades fast when value of n_p increases.

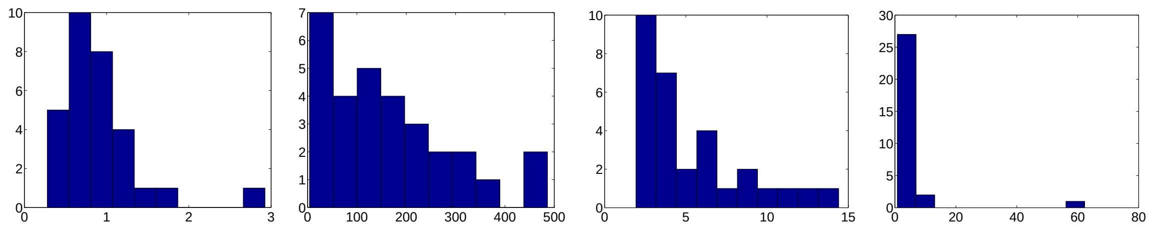
Table-4.19 reports the performance improvement of *SpiralRNN* with more training data, as the value of n_t increasing from 10 to 40. *Fig-4.18* shows the histograms of ε values of *SpiralRNN* in different n_t training rounds, where the model has the network size of 300 weights and the task is to reproduce $n_p = 5$ trajectory patterns. The initialization scheme-2 for the p_0 value has proven efficient, since results of all tasks at $n_t = 20$ training rounds have shown lower evaluation error than the threshold value (refer to *table-4.19*). An example of such an improvement can be found in histograms in *fig-4.18*, where the improvement from $n_t = 10$ (sub-figure *4.18(a)*) to

network size	100 weights				200 weights			300 weights	
n_t \ n_p	2	3	4	5	4	5	6	5	6
10	**-2.70**	-2.31	-	-	-2.15	-	-	-2.02	-
30	-3.28	**-2.75**	-2.22	-1.97	**-2.74**	-2.28	-	-2.50	-
50	-3.69	-2.93	-2.35	-2.14	-3.19	-2.29	-1.65	**-2.63**	-2.09
70	-3.64	-2.99	-2.44	-2.21	-3.31	-2.41	-1.75	-2.87	-2.21
90	-	-	-2.49	-2.23	-	**-2.66**	-1.88	-	-2.29
110	-	-	-	-	-	-	-1.93	-	-2.36

Table 4.17: *Performance improvement of* SpiralRNN *in successive training rounds with different tasks (w.r.t. network size and number of trajectory patterns n_p). Symbol "-" means that no simulation was performed for that scenario. Note that initialization scheme-1 is used and iall values are on a logarithmic scale.*

Model	100 weights			200 weights			300 weights	
	n_p=2	n_p=3	n_p=4	n_p=4	n_p=5	n_p=6	n_p=5	n_p=6
SpiralRNN	-3.23	-2.84	-2.44	-3.04	-2.76	-2.53	-2.97	-2.73
ESN	-1.15	-0.82	-0.66	-0.77	-0.64	-0.50	-0.70	-0.57
BDRNN	-1.52	-2.17	-1.98	-2.25	-2.24	-1.97	-2.27	-2.23
MLP	-2.11	-2.01	-1.10	-2.27	-1.73	-1.29	-1.76	-1.51

Table 4.18: *Comparison of evaluation error after $n_t = 20$ training rounds between models in different network sizes and in different tasks. Note that initialization scheme-2 is used and all values are on a logarithmic scale.*



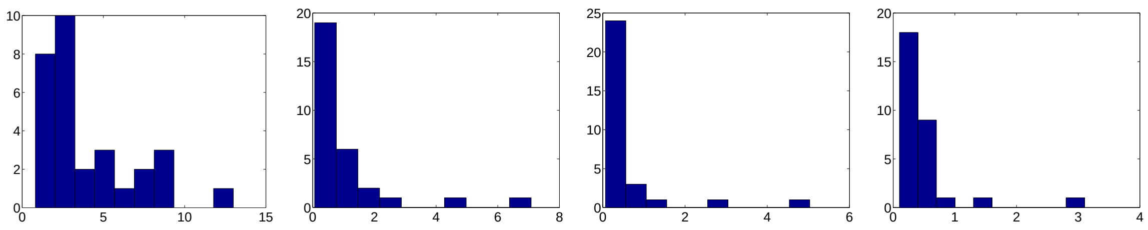
(a) *SpiralRNN* (b) *ESN* (c) *MLP* (d) *BDRNN*

Figure 4.17: *Histograms of evaluation error ε (over 30 simulations) of neural network models with 200 network parameters with $n_p = 4$ trajectory patterns and $n_t = 20$ training rounds. The X-axis is the evaluation error in 10^{-3} units, and the Y-axis shows the occurrence frequency of the respective ε value. Initialization scheme-2 is used. Note the different scales of the X-axis and the Y-axis in different sub-figures.*

$n_t = 20$ (sub-figure *4.18(b)*) is significant, which also implies that the model converges after $n_t = 10$ and before $n_t = 20$.

training rounds	100 weights			200 weights			300 weights	
	n_p=2	n_p=3	n_p=4	n_p=4	n_p=5	n_p=6	n_p=5	n_p=6
n_t=10	-2.76	-2.46	-1.99	-2.47	-2.23	-2.03	-2.39	-2.02
n_t=20	-3.23	-2.84	-2.44	-3.04	-2.76	-2.53	-2.97	-2.73
n_t=30	-3.43	-3.00	-2.54	-3.19	-2.99	-2.76	-3.17	-3.04
n_t=40	-3.63	-3.04	-2.63	-3.34	-3.11	-2.84	-3.31	-3.21

Table 4.19: *Comparison of the evaluation error ε of* SpiralRNN *models in different network sizes in different tasks. Note that all values are on a logarithmic scale and that initialization scheme-2 has been used.*



(a) $n_t = 10$ (b) $n_t = 20$ (c) $n_t = 30$ (d) $n_t = 40$

Figure 4.18: *Histograms of evaluation error ε (over 30 simulations) of the* SpiralRNN *model with 300 network parameters with $n_p = 5$ trajectory patterns at different n_t training rounds. The X-axis is the evaluation error in 10^{-3} unit, and the Y-axis shows the occurrence frequency of the respective ε value. Initialization scheme-2 is in use. Note the different scales of the X-axis and the Y-axis in different sub-figures.*

Discussion

Results shown in the above tables and figures clearly show that:

(1) Different trajectories can be stored by recurrent neural networks simultaneously. A well trained model can even distinguish trajectories which are not totally different right from the beginning, *e.g.*, the trajectories of products C and D shown in *fig-4.15*. Typical examples of trajectory prediction provided by a *SpiralRNN* model are given in *fig-4.19*, where the *SpiralRNN* model has 200 weights and reproduces all $n_p = 4$ trajectory patterns after $n_t = 20$ training rounds. Note that in each sub-plot the output trajectory is the autonomous result of the *SpiralRNN* model, given initial starting value x_0 but no further data input. This autonomous prediction does not necessarily match at every prediction step, but predictions of trajectories follow the trend of the respective target and are themselves distinguishable from each other (refer to *fig-4.19*).

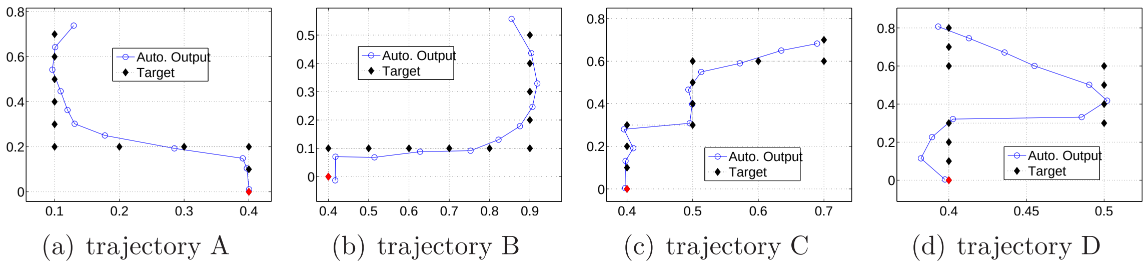


Figure 4.19: *The autonomous outputs on* $n_p = 4$ *trajectory patterns from a* SpiralRNN *model with 200 weights after* $n_t = 20$ *training rounds. The red dot at location* $(0.4, 0)$ *marks the warehouse entrance. Note that these are the autonomous outputs when a trained* SpiralRNN *model is only given different respective initial starting values as initial input. Initialization scheme-2 is in use.*

(2) Different neural network models have shown different convergence speeds, where the *SpiralRNN* model outperforms other approaches in different prediction tasks and with different network size.

It is worth mentioning that the *SRN* model shows similar or even slightly better results than the *SpiralRNN* but sometimes suffers from instability such that training fails completely (refer to *section-4.2.1*). On the other hand, this result indicates that the constraints on the hidden layer structure of a *SpiralRNN* do not severely constrain the modeling power of this architecture compared to the unconstrained *SRN*. The fact that *ESN* models don't provide acceptable performance shows that their reduced variability (only the output weights can be trained) imposes too many constraints and that the linear mapping imposed by the output weights is not sufficient for the conditional prediction described in this section. The *BDRNN* and *MLP* models have achieved better results than the *ESN*, but suffer from slow convergence. These results are also confirmed by the histograms in *fig-4.16* and *fig-4.17*.

(3) Simulations with initialization scheme-2, where p_0 is randomly initialized but kept constant afterwards for each trajectory pattern, have shown better performance than those with scheme-1 where p_0 is reset to zero without exception.

It is manifested in *table-4.19* that, after $n_t = 20$ or even $n_t = 10$ training rounds, the *SpiralRNN* model is able to achieve an evaluation error smaller or close to the threshold θ in all mentioned tasks. However, the results in *table-4.17* with initialization scheme-1 do not reflect such efficiency. *Fig-4.18*, as an example, shows the fast convergence of scheme-2 as well as of the *SpiralRNN* model.

Simulations in this section were conduced with trajectory data of dimension 2, but it can be easily extended to support the application with high dimensional data. The length of data pattern can vary, but a longer data pattern requires more training or a larger network for recognition and modeling of the dynamics of the whole trajectory pattern.

4.5 Using Online Learning of *SpiralRNN* in NN5 Data Prediction Competition

NN5 competition[20] is one of the leading competitions with an emphasis on utilizing computational intelligence methods in data prediction. The data in question come from the amount of money withdrawn from ATM machines across England. These data exhibit strong periodical (*e.g.* weekly, seasonally and yearly) behavior (ref. *fig-4.20*). The associated processes have deterministic and stochastic components. In general, they will not be stationary, as for example more tourists are visiting this area or a new shopping mall has opened. There are in total 111 time series in the database, with each time series representing the withdrawal from one ATM machine and each data point in particular time series indicating the withdrawal amount of the day from the particular ATM machine. All 111 time series were recorded from the 18th March 1996 till 22nd March 1998, and therefore contain 735 data points. The task of the competition is to predict the withdrawal of each ATM machine from 23rd March 1998 to 17 May 1998, 56 data points in total. The evaluation of prediction performance is based on the so-called SMAPE error value defined in *eq-(4.17)*, where y_t^* and F_t^* are respectively the predicted output and data.

$$E_{smape} = 1/n \sum_t^n \frac{|y_t^* - F_t^*|}{(y_t^* + F_t^*)/2} \times 100\% \tag{4.17}$$

4.5.1 Towards NN5 Competition

Theoretically, *SpiralRNN* can learn the dynamic characters of given data by itself. However, additional input can help to find a more accurate solution and to speed-up convergence. For the current task these are: (1) providing periodic input mimicking calendar information; (2) using committee of experts approach on top of neural network training.

[20]http://www.neural-forecasting-competition.com/

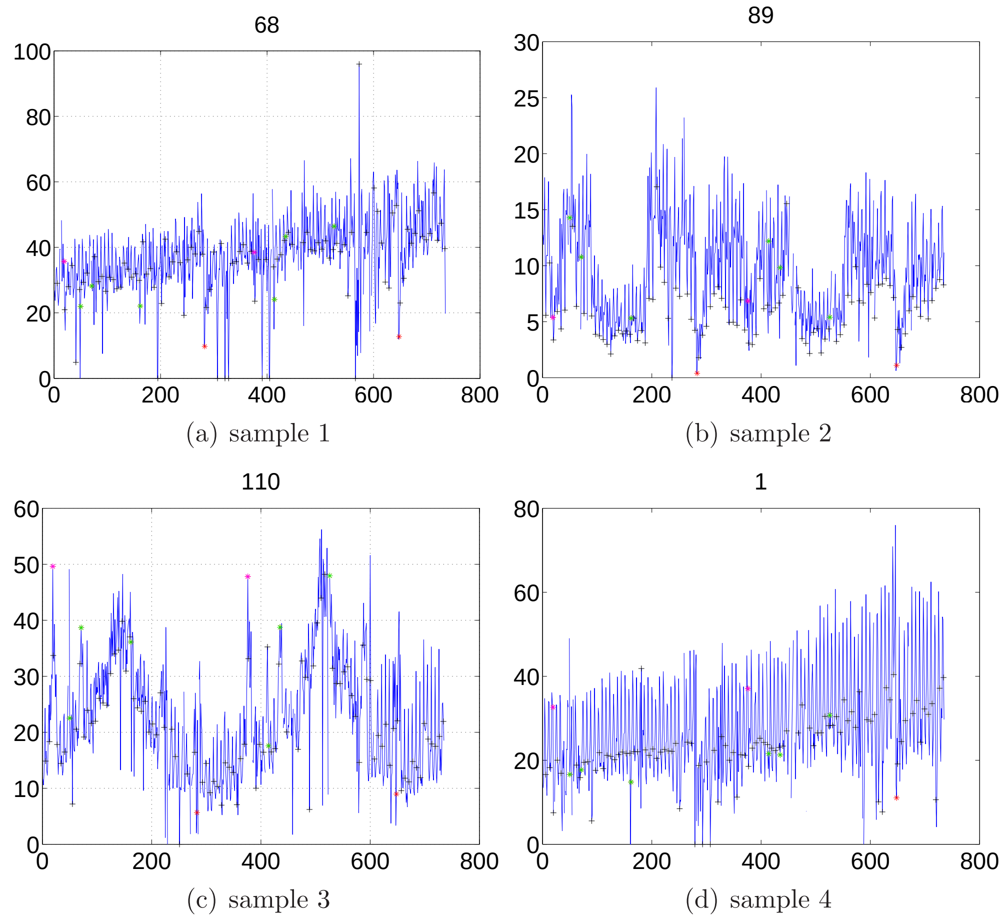


Figure 4.20: *Sample data from NN5 competition dataset with cross markers indicating Saturdays.*

Data characteristics

The time series data in the NN5 dataset exhibit at least the following features [21]; :

- $\mathbb{F}_1$ Strong weekly periodic behaviour dominates the frequency spectrum, usually with higher values on Thursday and/or Friday;
- $\mathbb{F}_2$ Important holidays such as the Christmas holidays (including the New Year holiday) and the Easter holidays have a visible impact on the data;
- $\mathbb{F}_3$ Several of the time series such as time series No. 93 No.89 show strong seasonal behavior, i.e. a yearly period;
- $\mathbb{F}_4$ Some of the time series (like No. 26 and No. 48) show a sudden change in their statistics, e.g. a shift in the mean value.

[21]Note that the Easter Fridays in 1996 to 1998 should have the indice “19”, “376” and “753” in the given data and the Saturdays before Christmas day of 1996 and 1997 have the indices “283” and “648”.

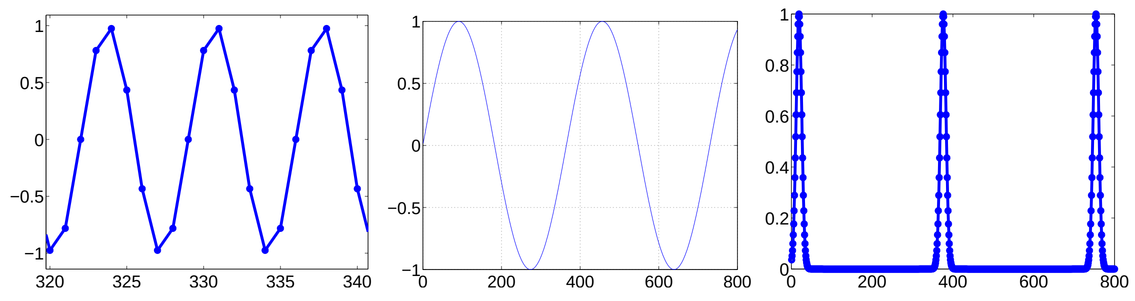
Pre-processing and additional inputs

The data presented to the neural network are mapped to a useful range by the logarithm function. In order to avoid singularities due to original zero values, they are replaced with small positive random values.

Additional sinusoidal inputs are provided as a representation of calendar information. These additional inputs include:

1. Weekly behavior addressing feature $\mathbb{F}_1$. Refer to *fig-4.21(a)* and note the period is equal to 7.
2. Christmas and seasonal behavior addressing feature $\mathbb{F}_2$ and $\mathbb{F}_3$. It is often observed from the dataset that, right after the Christmas holiday, withdrawal of money is low and then increases during the year, finally reaching its summit value right before Christmas. Seasonal features do not prevail in the dataset, but they do exist in several of them, *e.g.* time series No. 9, No. 88. As both are regular features with a yearly period, it makes sense to provide an additional input as shown in *fig-4.21(b)* which has the period value 365.
3. Easter holiday bump addressing feature $\mathbb{F}_2$. The Easter holidays did not have as much impact on the data dynamics as the Christmas holidays did, but it shows an effect on the usage of ATM in some areas (shown in some time series). Furthermore, as the 58-step prediction interval includes the Easter holidays of year 1998, the prediction over the holiday can be improved when the related data behavior is learnt. This additional input uses the Gaussian-distribution-shape curve to emulate the Easter holiday bump as in *fig-4.21(c)*.

Supplied with additional inputs, *SpiralRNN*s were trained online (see *fig-4.22*) such that data points were fed-in the network one-by-one and network parameters were trained between the time-step interval, as described in *section-3.2.2*.



(a) Weekly-input (b) Christmas-input (c) Easter-input

Figure 4.21: *Additional inputs of neural networks. On the X-axis is the time steps, and Y-axis is the additional input value.*

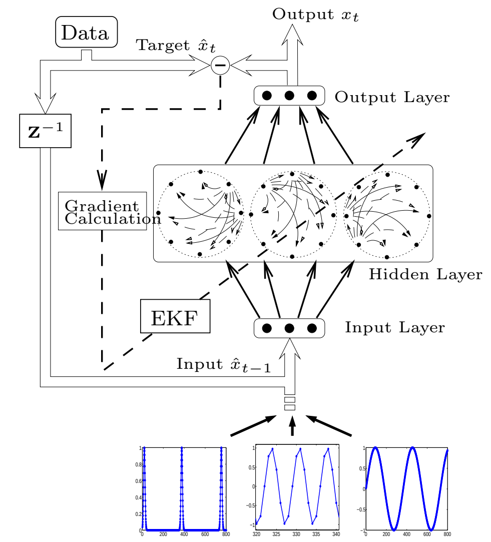


Figure 4.22: *Online training of* SpiralRNN *in NN5 competition with additional inputs.*

Committee of experts

SpiralRNN is capable of learning time series prediction with fast convergence; nevertheless, the learned weights correspond to local minimima of the error landscape as mentioned in [87]. As computational complexity is not an issue for this competition, a committee of experts ansatz is applied.

The committee of experts consists of several *SpiralRNN* models with identical structure but different initialization of parameter values. Each *SpiralRNN* model operates in parallel without any interference to the others. Online training will be done through the time series, step by step. The importance of experts' output in the committee, *i.e.* the weights of the experts' output, is determined by the average SMAPE value over the last 56 steps. Therefore, at time step $t = 735 - 56 = 679$, each model produces a prediction for the next 56 steps using its output as the input for next time step. After that, online learning continues until the end. For the predicted 56 values, the SMAPE error values compared to their respective data are determined and averaged over these values. Note that, during the autonomous prediction, current output of network will be fed back as input for next time step, therefor there is no intervention from outside of neural network. This autonomous output, namely $\vec{U}_t$, will be stored and SMAPE error value will be calculated by comparing $\vec{U}_t$ with the future available data values in the time series. Each *SpiralRNN* model k measures the average SMAPE value ε_k according to *eq-4.18*, where E_s refers to the SMAPE error calculation.

$$\varepsilon_k = \frac{1}{56} \sum_{t=1}^{56} E_s(\vec{U}_t, \hat{U}_t) \tag{4.18}$$

Similarly, autonomous predictoin will also be produced at the last time step $t = 735$ where data $\hat{y}_t|t = 735$ is provided to the network model k and generates the 56-step autonomous output P_k. With the SMAPE error measurement ε_k and the 56-step autonomous prediction P_k, one can produce the final prediction using the voting weights of each expert k, such that:

$$\phi = \sum_k^n 1/\varepsilon_k^2 \tag{4.19}$$

$$P_t = \frac{1}{\phi}\sum_{k=1}^n P_{t,k}/\varepsilon_k^2 \tag{4.20}$$

The whole procedure is listed in *table-4.20*, and a schematic diagram *fig-4.23* depicts the organization of the committee.

0.	Initialize the n experts;
1.	For a *SpiralRNN* model k, implement on-line training with the data and make a 56-step prediction U_k at time step $t = 679$. The prediction value U_k will be compared to the data in order to obtain the average SMAPE error value according to *eq-4.18*.
2.	After the prediction at time step $t = 679$, continue the online training till the end, and produce another 56-step autonomous prediction P_k.
3.	Based on their ε_k values, combine the prediction P_k according to *eq-4.20*.

Table 4.20: *Committee of experts.*

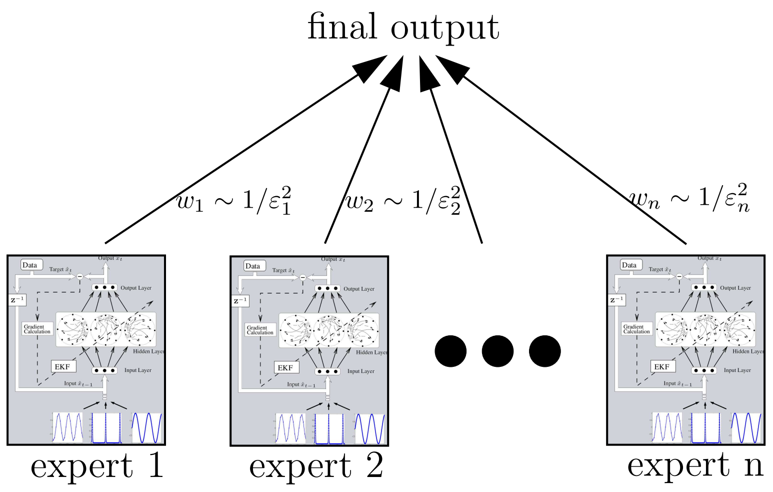


Figure 4.23: *Committee of* SpiralRNN *experts where their weights in committee are determined by their SMAPE values on testing dataset.*

Adapted EKF on-line training of *SpiralRNN*

The *SpiralRNN* model used for the competition is constructed according to the description in *chapter-3* and in *section-4.1*. Modifications locate in (1) the number of output is fixed to one so that there is no need to predict the additional inputs whose values are known beforehand and are easy to obtain, and (2) the adapted calculation of gradient matrix Ψ in *eq-3.43*.

Recall that logarithm operator is used in order to map the data into a reasonable range before they are fed into the neural network, and that SMAPE error evaluation is used as in *eq-(4.17)*. Therefore, the calculation of gradient Ψ is adapted as in *eq-4.23* with $y_t = ln(y_t^*)$ being the output in transformed scale and $F_t = ln(F_t^*)$ the data in transformed scale:

$$s = \exp(y_t) + \exp(F_t) \tag{4.21}$$

$$d = \exp(y_t) - \exp(F_t) \tag{4.22}$$

$$\Psi_t = -\frac{\exp(y_t)}{s}\left(sign(d) + \frac{|d|}{s}\right)\frac{\partial y_t}{\partial w_t} \tag{4.23}$$

As there exist data with missing values, the parameter-update is skipped when missing data are involved in the training, but meanwhile still accumulate the gradient of output *w.r.t.* parameters, until data values are available again.

4.5.2 Results

Here, some results of the prediction task are shown. In order to provide evaluation results, the last 56 data of each time series were treated as test data while the remaining data were used for online-training. In *fig-4.24*, prediction and data are compared for time series 35 which has a pronounced weekly periodicity. Obviously, this periodicity can be reproduced very well, even details like the small bump.

Seasonal behavior of data can also be learned as shown in *fig-4.25*. The curves in both sub-plots begin with the values in Christmas holidays (with time indices around 280 and 650). The rise and fall of the data during the first about 100 days and a subsequent rise in *fig-4.25(a)* can also be seen one year later in *Fig-4.25(b)*. Obviously, the model is able to capture this behaviour pretty well.

Easter holidays can also be recognized by the trained model as shown in *fig-4.26* for time series No. 110. Since the expert-weight in the committee is determined by the average SMAPE value as in *eq-4.20* which doesn't focus on the error during Easter holiday period, the ability to predict the accurate values for the easter holidays has been diluted. But as shown in *fig-4.26*, the jumping feature in data for Easter holiday has been recognized by the on-line training. The Easter holidays in 1996 to 1998 are indexed at times around

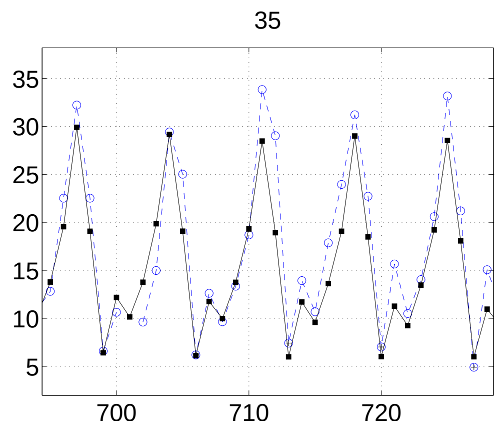


Figure 4.24: *Comparison between result and data showing weekly behavior for time series 35. Dashed line with circles is the data and solid line with squares is the prediction. Time is on the X-axis.*

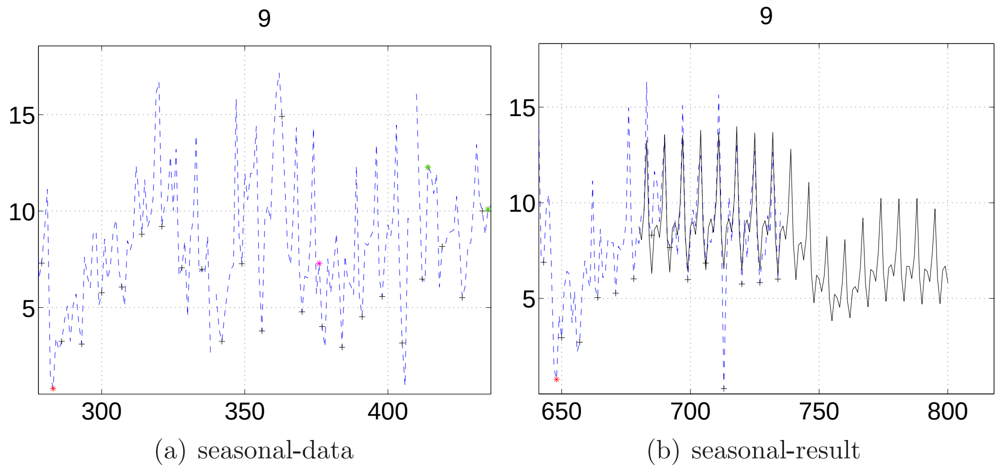


Figure 4.25: *Comparison between result and data for time series 9 showing seasonal behavior. Dashed curve is the data and solid curve is the prediction. X-axis is time.*

20, 375 and 755. In *fig-4.26*, the prediction for the Easter holidays 1998 shows a similar behaviour as the Easter holidays in 1996 and 1997 with a pronounced peak.

Table-4.21 shows the SMAPE errors on the test set (*i.e.* the data from the last 56 time steps) with a varying number of experts. Obviously, the number of experts does not alter the average result, which allows to save computational effort. *Fig-4.27* shows the histogram of committee-averaged SMAPE values over 111 time series in the dataset. The majority of the results have a SMAPE value around 20. It is shown that, using the committee of experts approach where the weight of each expert's vote is determined by the error of autonomous output, one can avoid the degradation from the over-fitting of some experts.

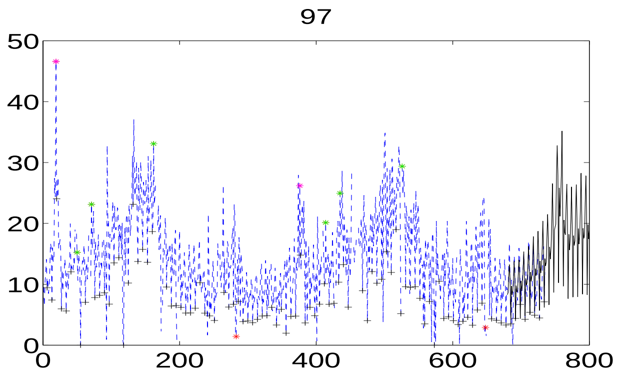


Figure 4.26: *Prediction result (solid curve) and data (dashed curve) for time series 110 showing a peak around Easter. X-axis is time.*

# experts	3	5	10	15	20	30
SMAPE	21.44	20.92	21.41	22.26	22.18	21.6

Table 4.21: *Statistic results. The committee-averaged SMAPE error value of the expert committee on all* 111 *time series with different number of experts.*

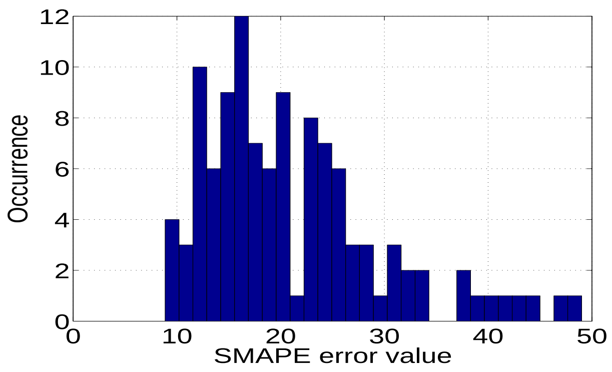


Figure 4.27: *Histogram (over* 111 *time series) of committee-average SMAPE error from experts .*

4.6 Analysis on *SpiralRNN*s

This section makes a general analysis of *SpiralRNN* models, using some of the results and methods from previous sections. Topics include the attractors learnt by the *Spiral-*

RNN model, the short-term memory capacity and the associative memory capacity of a *SpiralRNN* model.

4.6.1 Stability of Attractors

Stability of a trained *SpiralRNN* model has been demonstrated by autonomous tests with deviated initial input values. Note that, in autonomous tests, the model uses the previous output value as the current input value. The only exception is the initial input value which is supplied from outside.

Fig-4.28 shows the autonomous outputs generated by a trained *SpiralRNN* model from the simulation with data set *Spike21*. Each sub-figure in *fig-4.28* shows the result from simulation with different initial input values $\{\pm 0.1, \pm 1, \pm 10, \pm 100\}$ respectively. The first part of the autonomous output corresponds to the transition phase from the initial state to the attractor regime. Without exception, the models with different initial input have shown periodical behaviour in the attractor regime, where the autonomous outputs have the period equal to 21.

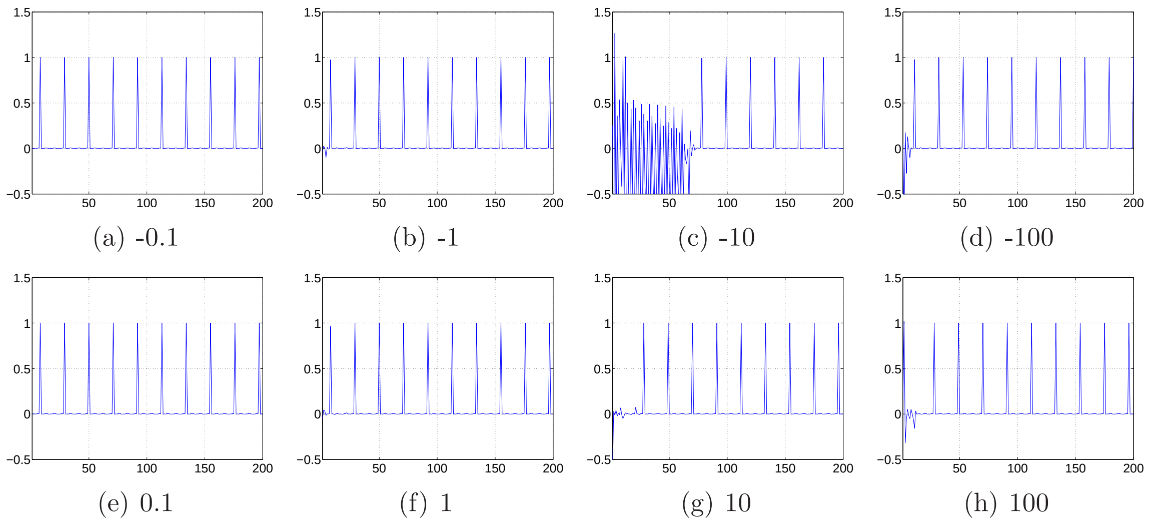
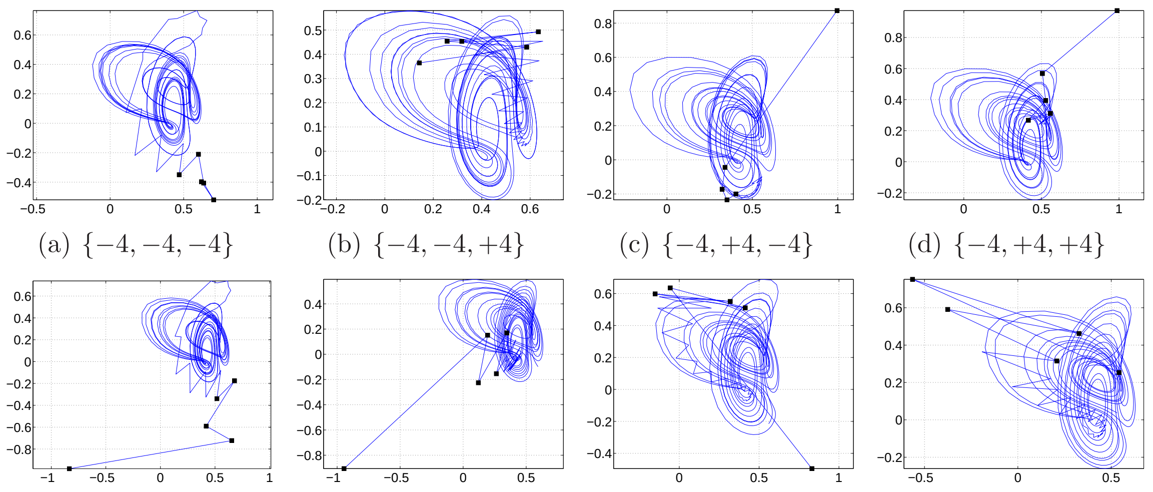


Figure 4.28: *Stability test of a trained* SpiralRNN *model from the simulation with the Spike21 data set. The sub-figures show the results for different initial input values* $\{-0.1, -1, -10, -100, +0.1, +1, +10, +100\}$ *respectively. The X-axis is the time in the autonomous tests, and the Y-axis is the output.*

As another example, *fig-4.29* depicts the trajectories of two hidden-neuron activations during the autonomous test with a trained *SpiralRNN* model from a simulation with the *Lorenz* data set. In the autonomous test, the *SpiralRNN* model, with a different initial input value, iterates itself for 1000 time steps. Meanwhile, the hidden-state vector is recorded for each time step. Sub-figures in *fig-4.29* show the trajectories of the first and second entries of the hidden-state vector. Positions of two entries in the first five time steps of the autonomous test are denoted by black square markers. Depending on the initial position, the trajectories follows different zigzag paths but are always attracted by the limit cycle driven by the *Lorenz* data. Note the different scales of the sub-figures, and that the position of the limit-cycle is in the range $[-0.2, 0.6]$ for the X-axis and $[-0.2, 0.6]$ for the Y-axis in all sub-figures[22].



(a) $\{-4, -4, -4\}$ (b) $\{-4, -4, +4\}$ (c) $\{-4, +4, -4\}$ (d) $\{-4, +4, +4\}$

(e) $\{+4, -4, -4\}$ (f) $\{+4, -4, +4\}$ (g) $\{+4, +4, -4\}$ (h) $\{+4, +4, +4\}$

Figure 4.29: *Stability test of a trained* SpiralRNN *model from simulations with the Lorenz data set. Each sub-figure shows the trajectory of two hidden neurons during autonomous tests with different initial inputs. The starting point coordinates are given beneath each sub-figure. The X-axis and the Y-axis respectively refer to activation values of these two hidden neurons. The first five positions of the trajectory are denoted by black square marks.*

4.6.2 Short-Term Memory Capacity

The short-term memory $\mathscr{M}_{st}$ of a *SpiralRNN* model has been evaluated with the *Spike* time series. *Spike* time series are those periodical time series, where most of the entries have a constant value and periodical sparse entries having an other constant value (refer to *section-4.1*). The distance between two spikes is called the period $\mathscr{P}_t$ of the time series. The autonomous test with *Spike* time series requires a model being able to retain the spike information from the input and able to produce repetitions of spikes in output in an autonomous manner. Note that, during the autonomous test, there are no further data affects the procedure except for the initial input. Therefore, if a network is able to produce

[22] Trajectories of different-hidden neurons can have different shapes of limit cycle and position.

an autonomous output of spikes, one can claim that the trained network has a short-term memory capacity $\mathscr{M}_{st}$ of not less than the period of the time series.

In each simulation, following the network setting (with $\gamma = 1$) as described in *section-4.1.2*, a *SpiralRNN* model with $\mathbf{N}_h$ hidden nodes is trained with the *Spike* data set with the period value $\mathscr{P}_t$. After every 2000 training steps, model generates an autonomous output $\hat{Y}$ of length[23] 200. If the autonomous output $\vec{Y}$ and the respective target $\hat{Y}$ satisfy *eq-(4.24)*, it is claimed that this network has the short-term memory $\mathscr{M}_{st} \geq \mathscr{P}_t$, then the current simulation is terminated and a new simulation will be implemented with same network structure but the training data is replaced with the *Spike* time series of longer period (*i.e.* larger $\mathscr{P}_t$ value). Otherwise, the training is continued, until the model has been trained for $1e5$ steps. Such $1e5$-step simulation will be implemented at most 30 runs. If the model has not satisfied *eq-(4.24)* in any of the 30 simulations, the network is claimed to have the shor-term memory $\mathscr{M}_{st} < \mathscr{P}_t$, and a new simulation is implemented with same training data but larger network (*i.e.* larger $\mathbf{N}_h$ value). Note that both values of $\mathscr{P}_t$ and $\mathbf{N}_h$ start from 2, and increase with the step pace 2.

$$\log_{10}(\hat{Y}_i - Y_i)^2 < -2, \quad \text{for all } i \in [1, 200] \tag{4.24}$$

The assessment result of short-term memory capacity of *SpiralRNN* models is shown in *fig-4.30*. The short-term memory capacity of *SpiralRNN* models increases as the network become larger, whilst the increment is getting smaller. The relationship is approximated[24] by *eq-(4.25)*, where the margin errors are calculated with the confidence level 95%. The term "$-2 + |b|$" in *eq-(4.25)* is used in order to avoid the complex value when $c < 1$.

$$\begin{aligned} \mathscr{M}_{st} &= a(\mathbf{N}_h - 2 + |b|)^c, \quad \mathbf{N}_h \geq 2 \\ \text{with } a &= 16.8 \pm 0.88, \quad b = 0 \pm 0.66, \quad c = 0.42 \pm 0.10 \end{aligned} \tag{4.25}$$

Recall that the *ESN* model and the *BDRNN* model manually set the maximum absolute eigenvalue of hidden-weight matrix W_{hid} to a particular value $\lambda_{max} < 1$, in order to enable the "echo effect" (*i.e.* the amplitude of the hidden-state vector is shrinking when model iterates). In *SpiralRNN*, the parameter γ (ref. *eq-(3.7)* and *eq-(3.8)*) has a similar functionality. However, the γ value doesn't establish a rigid limit on the eigenvalue spectrum of matrix W_{hid}, but reflects the difficulty of model to achieve a wider eigenvalue spectrum. Taking the γ value from the set[25] $\left\{1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\right\}$ respectively, *SpiralRNN*s have produced different performance in terms of short-term memory capacity. *Fig-4.31* depicts the result, where each curve denoted by the same type of markers represents the $\mathscr{M}_{st}$ values of model

[23] The length of autonomous output has to be greater than the period of the training data, such that at least one spike is supposed to show up in the autonomous test.

[24] The regression value of variable b in *eq-(4.25)* is not exactly equal to 0, but 0.0061.

[25] Simulations with γ value bigger than 1, such as 2 and 4, have been considered, but they all reported instability problems since system becomes instable with too large eigenvalue of matrix W_{hid}.

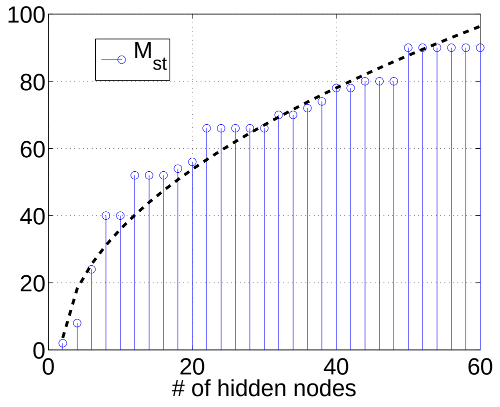


Figure 4.30: *The short-term Memory capacity $\mathscr{M}_{st}$ v.s. the amount of hidden neurons $\mathbf{N}_h$ of* SpiralRNN *model. Circles are from assessment, and the dashed curve is the regression curve drawn by eq-(4.25).*

with particular γ value. The dashed curves are drawn from the regression results based on the assessment values denoted by markers, where the blue curve is the regression result for the $\gamma = 1$ case and the red curve for $\gamma = 1/2$ (The parameter values when $\gamma = 1/2$ are $a = 7.9$, $b = 0.03$ and $c = 0.44$.).

It can be observed from the figure, that:

(1) With a bigger γ value, *SpiralRNN*s having the same $\mathbf{N}_h$ hidden nodes will possess larger short-term memory capacity.

(2) For larger γ values, *e.g.* $\gamma = 1$ or $\gamma = 1/2$, the short-term memory of *SpiralRNN*s satisfies the power law $\mathscr{M}_{st} = \alpha(\mathbf{N}_h - n_0)^{c_0}$, where the $n_0 \simeq 2$ and $c_0 \simeq \frac{1}{2}$, and the α value is proportioned to the predefined value of γ.

(3) The $\mathscr{M}_{st}$ curves for small γ values, *e.g.* $\gamma = 1/4$ and $\gamma = 1/8$, have shown threshold behavior. After the threshold phase at the beginning, the latter part of both curves seem to start satisfying a power law. The values of both curves are also roughly proportioned to their respective γ values.

Therefore, increasing the γ value can extend the short-term memory capacity of *Spiral-RNN*s. However, as mentioned, the model will become instable when the eigenvalue spectrum of matrix W_{hid} is recklessly enlarged, choosing a moderate γ value is therefore a tricky job. Practically, a typical setting is $\gamma = 1$.

Note that, even though a large γ value tends to widen the eigenvalue spectrum of hidden matrix, the actual eigenvalue spectrum is depending on both values of γ and $\tanh(\vec{\xi})$ (ref. *eq-(3.7)*) where the squashed function "tanh" helps the train of ξ value and obtains an appropriate eigenvalue spectrum.

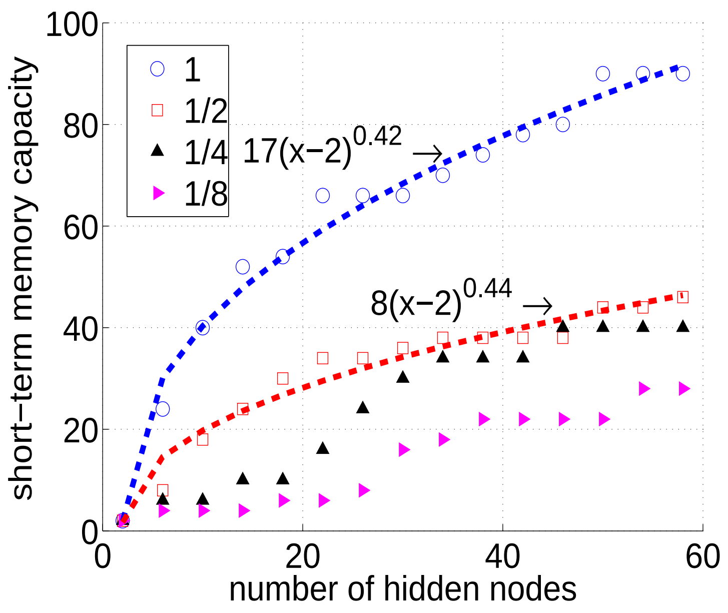


Figure 4.31: *Comparison of the impact of γ value on short-term memory of* SpiralRNN*s. The assessment data are obtained by repeating the simulation as described above, but setting the γ values to one of $\{1, , \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\}$ and changing the $\mathbf{N}_h$ increasing step-pace from 2 to 4. The regression equations are shown aside the regression curves, but the approximated coefficients are used for the sake of clarity.*

4.6.3 Associative Memory Capacity

Generally, the associative memory of a neural network model illustrates the neural network's ability of memorizing data patterns, where "data pattern" means one single data vector. The neural network is in such a case a static model, mapping different inputs to different instances in a finite set of target pattern. However, the associative memory of the network model in terms of temporal time series, instead of tracking down to attractors presented by training data, requires the network model to be able to store the dynamics of various trajectory patterns, burdening more difficulties on the learning model.

Conditional prediction applications in *section-4.4* has indicated that *SpiralRNN* has certain associative memory and is able to reproduce different trajectory patterns, given the initial starting vector. To evaluate the associative memory, *SpiralRNN*s configured with different numbers of hidden nodes $\mathbf{N}_h$ are trained and tested, where the simulation data consists of varying number n_p of trajectory patterns. Giving particular values of $\mathbf{N}_h$ and n_p, conditional prediction simulation will be carried out in a similar way as in *section-4.4*:

(1) The experiment starts with a smaller networks (The value of $\mathbf{N}_h$ starts from 6, such that the network comprises 3 hidden units and each hidden unit has 2 hidden nodes);

Training data is the combination of $n_p = 2$ trajectory patterns.

(2) With each pair of $\{\mathbf{N}_h, n_p\}$, at most 30 simulations are performed, each of which contains 50 training rounds. In each training round, the learning model is trained on-line with the presentation of these n_p trajectory patterns, in a shuffled sequence but each pattern only once. The hidden-state vector of network is reset before the training with any trajectory pattern starts (refer to *section-4.4*). After every 10 training rounds, a testing round starts, trying to reproduce all n_p trajectories in an autonomous manner separately.

(3) The autonomous output results are evaluated. Let l_p be the length of trajectory patterns (the l_p value is identical to all pattern), $x_{i,k}$ be the i^{th}-step ahead prediction over the k^{th} pattern, and $\hat{x}_{i,k}$ be the respective target, the evaluation error ε is calculated as:

$$\varepsilon = \frac{1}{n_p}\frac{1}{l_p}\sum_{i=1}^{l_p}\sum_{k=1}^{n_p}\left(\hat{x}_{i,k} - x_{i,k}\right)^2 \tag{4.26}$$

(4) If it satisfies $\varepsilon \leq 0.05^2$, this particular *SpiralRNN* network with $\mathbf{N}_h$ hidden nodes is claimed to be capable of reproducing n_p number of trajectory patterns of length l_p, and thus the associative memory satisfies $\mathscr{M}_a \geq n_p$. The model (with the same structure but re-initialized values) will then be assessed with a training data with more patterns, *i.e.* higher n_p value.

If $\varepsilon \leq 0.05^2$ is not satisfied at least once within 30 simulations, it is claimed that *SpiralRNN* with $\mathbf{N}_h$ hidden nodes is not able to reproduce n_p number of trajectory patterns with length l_p. New experiment will start with same value in n_p but a larger *SpiralRNN* model by increasing the $\mathbf{N}_h$ value [26] by 3.

Note that the threshold value is set to 0.05, half the grid distance 0.1, so that the prediction can be rounded to the nearest grid point if the error is less than 0.05.

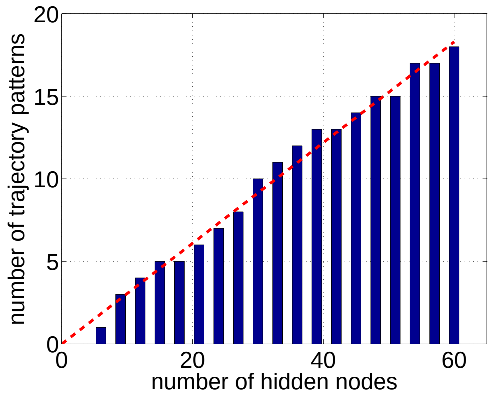
(5) The experiment keeps continuous until simulations with $\mathbf{N}_h = 60$ are finished.

The training data, *i.e.* combination of trajectory patterns, are generated in a fashion different from *section-4.4*. Each trajectory starts from the origin rather than $(0.4, 0)$, pace of trajectory is again 0.1 which means, at one step, one of the coordinates of trajectory is kept unchanged while the other one changes by 0.1. In each trajectory pattern, one coordinate of trajectory coordinates is monotonically increasing or decreasing. The initial starting point (including the additional information ϕ) of the data is uniformly randomized in the range $[-5, 5]$ for each pattern. A total of 60 random patterns is generated, where sample patterns in each simulation are randomly selected, rather than selecting them in sequence as in *section-4.4*.

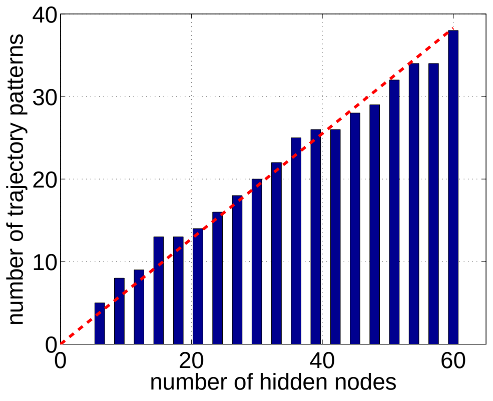
Fig-4.32 depict the assessments on the associative memory of dynamics. Each figure

[26] *i.e.* each hidden unit has one more hidden node

demonstrates the result with different value on l_p, the length[27] of trajectory pattern, where $l_p = 11$ in *fig-4.32(a)* and $l_p = 5$ in *fig-4.32(b)*.



(a) for longer pattern



(b) for shorter pattern

Figure 4.32: *The associative memory of a* SpiralRNN *shows the ability to store several dynamics of trajectory patterns. The X-axis is the network size of model indicated by the total number of hidden nodes, and the Y-axis is the maximum number of patterns which the model can learn. Red dashed lines are regression lines showing the linearity between* $\mathscr{M}_a$ *and* $\mathbf{N}_h$. *(a) Trajectory patterns of length* $l_p = 11$*; (b) trajectory patterns of length* $l_p = 5$.

It is easy to observe that the associative memory capacity $\mathscr{M}_a$ of *SpiralRNN*s is roughly linear with the amount of hidden nodes $\mathbf{N}_h$. The dashed red lines in both figures suggest this linearity. In particular, in *fig-4.32(a)* the associative memory satisfies *eq-(4.27)*, and in the case of short trajectories in *fig-4.32(b)* it satisfies *eq-(4.28)*. The reasons why the associative memory capacity with trajectory pattern of $l_p = 11$ is worse than that the one with $l_p = 5$ are:

- A longer trajectory pattern requires a longer training time to train the model so that it can adapt to the dynamics imposed by the data;
- The evaluation is based on the autonomous output where an error at one time step degenerates future results exponentially, therefore the evaluation error for longer trajectory pattern tends to be larger than that for shorter ones.

$$\mathscr{M}_a \simeq (0.31 \pm 0.02)\mathbf{N}_h \quad \text{(for longer patterns)} \tag{4.27}$$
$$\mathscr{M}_a \simeq (0.64 \pm 0.04)\mathbf{N}_h \quad \text{(for shorter patterns)} \tag{4.28}$$

The linearity between the network's associative memory capacity and the network's $\mathbf{N}_h$ value can also be found in Hopfield networks. According to [88], the associative memory

[27] The length of trajectory is counted here without taking the initial starting point into account.

capacity[28] $\mathscr{M}_{hf}$ of Hopfield network satisfies $\mathscr{M}_{hf} = 1/(2\log\epsilon^{-1})\mathbf{N}_h$ for small error ϵ, where ϵ is the threshold value of acceptable evaluation error. Even though it is unfair for *SpiralRNN*s to compare the recognition of trajectory pattern with the recognition of fixed point pattern, the associative memory of Hopfield networks at $\epsilon = 0.05$ reads: $\mathscr{M}_{hf} = 0.17\mathbf{N}_h$, which indicates the superiority of *SpiralRNN*s over Hopfield networks in associative memory capacity.

[28]The associative memory capacity of Hopfield networks refers to the number of fixed points in the hidden-state space.

Chapter 5

The *Duty-Cycle Reduction* and *Evolution Framework* for Distributed Sensor Networks

A distributed sensor network consists of sensor nodes. Among these nodes, there is no master sensor being in charge of data flow and central data processing, but all of these jobs have to be done locally by each sensor. This chapter addresses some issues in building such distributed sensor networks with data predicting ability, including the energy consumption, which is one of the paramount challenges for sensor network applications, and the prediction performance of distributive sensor nodes. Solutions are provided in two categories, namely *intra-node* viewpoint and *inter-node* viewpoints.

Intra-node

In the *intra-node* viewpoint, solution for energy consumption focuses on the duty cycle of sensor nodes. This can be done mainly within the sensor node and doesn't directly involve its neighbors.

The energy consumption of a sensor node can be greatly curtailed with the diminution of its duty cycle. Duty cycle of one device is defined as the proportion of time during which this device is operated. When a sensor is embedded with a well trained model which provides good prediction, it can diminish the effort to receive data from neighbor by substituting of the requesting communication with the prediction. This is also the reason why a learning model with good performance in long-term prediction is required. *Fig-5.1* has illustrated this concept.

Inter-node

In *inter-node* view-point, reduction in energy consumption can be realized by changing the embedded learning model of sensor, which requires the information from its neighbors.

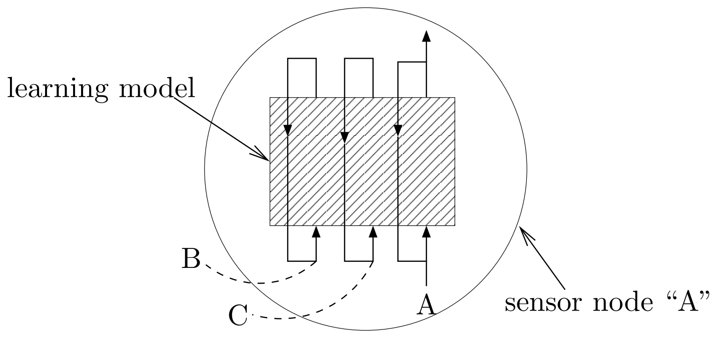


Figure 5.1: *Sketch of sensor “A” emphasizing the concept of the* duty-cycle reduction*. Shaded box in the middle represents the embedded learning model. Symbols “B,C” refer to the data from neighbor sensors “B,C” respectively. Symbol “A” refers to the measured data by sampling. Dashed arcs, indicating the expensive wireless communication, are replaced by the cheap prediction from sensor itself.*

To reduce the communication cost, one direct way is to change the interface of the embedded model. The interface of the embedded model refer to the combination of available data streams, which represent themselves as the model input (and output). By changing such interface, the embedded model has the chance to choose those “important” data streams as input and omit those “unimportant” data streams.

Furthermore, changing the embedded model applies not only to the interface but also to the inner structure of model itself, where such training of embedded model can be intensive. To reduce the energy consumption in computation, the structure of embedded model is simplified by evolution or co-evolution. This requires the sensor node to request the structure information from its neighbors.

Both situations mentioned above require the sensor to be “context-aware” to have have certain knowledge of the environment and its neighbor. Only based on such knowledge, it can takes the decision on choice of data streams and on structure modification.

Solutions from *intra-node* and *inter-node* view-points can be implemented separately, however, they are connected closely to each other. In one way, from *intra-node* viewpoint it requires each sensor to train a sufficient recurrent neural network (RNN) model for prediction, whereas a good prediction is based on the assumption that enough spatial data are available (ref. *chapter-1*), which can be influenced by *inter-node* operations. In the other way around, from *intra-node* view-point, the modification of model structure can degenerate the prediction performance, and requires some time for the “recover” of the model. This will consequently hinder the implementation of *intra-node* solutions. Furthermore, both *intra-node* and *inter-node* solutions are feasible under the assumption that the embedded model is fast-converge efficient on-line learning prediction model. Only with such embedded model, can the prediction be wealthy enough to replace the measured data, and can the model be able to recover from evolution in reasonable time.

In the following sections, details of the *duty-cycle reduction* and *evolution framework* will be given.

5.1 The *Duty-Cycle Reduction*

The idea behind the *duty-cycle reduction* scheme is straightforward. Duty cycle is the one of the main criteria to evaluate the life-time of a sensor node. When the duty cycle is reduce, the amount of working time of a sensor within unit time is reduced, hence the life time of the device with given limited energy is accordingly prolonged.

Recalling the typical working cycle of a TelosB sensor node in *fig-2.8*, communication normally occupies the main portion of the working time slots because of the administrative overhead, the waiting and listening time as well as the ever-strengthen processor[29]. Moreover, in a broadcasting radio network, sensor node usually spends more time in requesting communication (listening to neighbors) than in issuing communication (sending out data). Therefore, duty cycle can be greatly reduced if the frequency of requesting communication is lowered. To achieve this, replace the transferred data by their respective prediction data is one of the most efficient ways.

Even though such substitution of data streams can save energy spending, a long-standing absent of measured data from neighbors will cause the learning model diverge, and sometimes make the system unstable. The learning model need the genuine measured data, from time to time, in order to keep the model dynamics on the right attractor area. Therefore, the communication requesting for measured data need to be resumed periodically. Such period of activity pattern for requesting communication is determined by a positive integer $\mathscr{A}_p \in \mathbb{N}^+$ and is named the "activity period" (for requesting communication) of sensor node in this context. Being related but different from the duty-cycle of a sensor, the value of $\mathscr{A}_p$ refers to the number of working cycles of sensor, during one of which the requesting communication is allowed. *Fig-5.2* illustrates the concept of activity period.

By implementing the *duty-cycle reduction*, sensor node can directly save the energy spent on requesting communication. As mentioned, the advance of integrated circuit has made computation much more cheaper than that of communication, and such gap becomes larger. Therefore, the *duty-cycle reduction* scheme will have more and more superiority in the future.

The *pseudo*-code of the *duty-cycle reduction* is given in *table-5.1*, where the requesting communication operation is implemented only when "noReduction=1". Note that, even

[29] Faster processor (data processing capability) relatively makes the communication much more expensive.

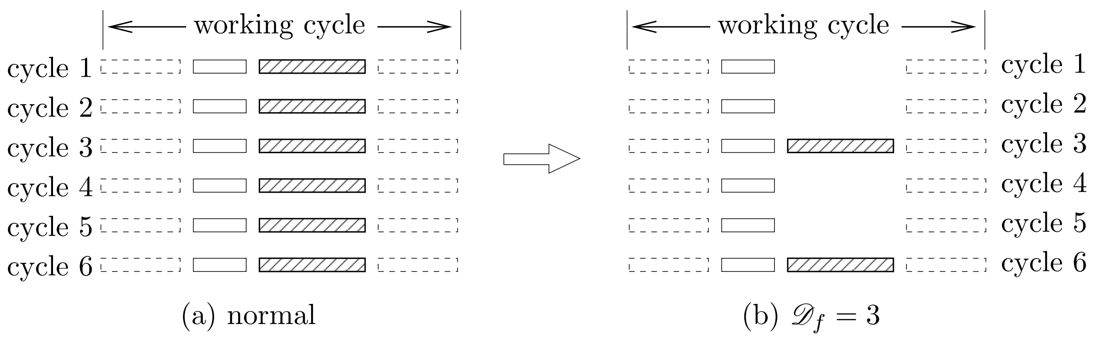


Figure 5.2: *Explanation of activity period. Each working cycle consists of the time slot for sending (solid bar without shadow), the time slot for receiving (solid bar with shadow), and time slots for other operations or sleeping (bars with dashed line). Each sub-figure lists distribution of time slots for* 6 *successive working cycles. (a) the normal case; (b) the activity period value* $\mathscr{A}_p = 3$*: in every 3 successive working cycles, sensor is allowed to receive data from outside only once.*

during working cycles where requesting communication is omitted, the on-line training of the embedded model is kept continuous, because the sensor can always have access to the real data of its own measurement[30]. In this case, the calculation of gradient only takes the residual value, corresponding to sensor's own measured data, into account.

5.2 The *Evolution Framework*

Because of the limit on the memory capacity and the processing speed of a processor, standard evolutionary algorithm based on genotype selection from large population cannot be applied in sensor network applications. On the other hand, sensor network consists of large amount of sensor nodes, and they can exchange data via the wireless communication. In this sense, evolution can be realized, if sensors are able to deliver their component (*i.e.* model structure) information to their neighbors, and adopt information from neighbors. However, conventional recurrent neural networks (RNN) cannot be easily adapted to such co-evolution concept in sensor network, either because of the massive-junction of the hidden layer (*e.q. ESN*, SRN *etc.*) or the identicalness of structures between separated units (*e.g. BDRNN*). The *SpiralRNN* model, on the contrary, can be easily split into hidden units because of its special structure. Taking hidden units as samples in the population, sensor nodes exchange structure information among each other, and try to find the better combination of hidden units to form the embedded model, as depicted in *fig-5.3.* Note that only one of evolutionary operations such as "discard" and "absorption" will be implemented at each time.

[30] Depending on applications, energy spent in sampling is normally cheaper than in communication.

```
for each time step
    if time step is multiple of $\mathscr{A}_p$, noReduction=1; else, doReduction=1; end

    if noReduction=1
        request data from neighbors;
        replace input by transferred data plus own measurement data;
    elseif doReduction=1 (duty-cycle reduction)
        replace input by previous output value plus own measurement data;
    end
    iterate the SpiralRNN model and generate output result;
    if noReduction=1
        calculate the residual of output comparing with target;
    elseif doReduction=1 (duty-cycle reduction)
        calculate the residual of the output of neuron
        which is corresponding to its own measurement data;
    end

    implement training of parameters based on the residual value;
end
```

Table 5.1: *Pseudo code for the* duty-cycle reduction*.*

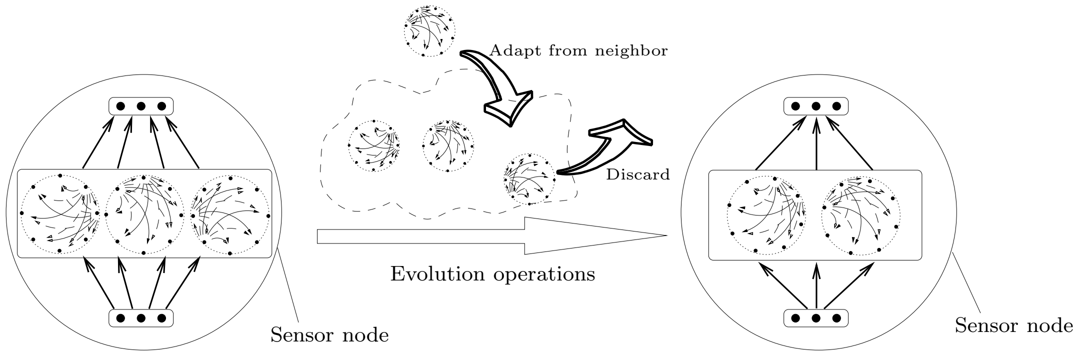


Figure 5.3: *Structure modification in a sensor node. The embedded model of sensor node is changed by re-combining the hidden units from itself and from neighbors. Note that only one of evolutionary operations such as "discard" and "absorption" will be implemented at each time.*

In addition to modifying the model structure itself, evolution should also be able to modify the model interface and find a better combination of input data streams. The model interface refers to the input and output of the embedded model. As the ultra goal of the model is to predict the local measured data, rather than those measured data from the neighbors. The embedded model can theoretically choose any combination of measured data from neighbors plus the measured data from itself, and use this combined data in the training of model parameters. Such evolution not only simplifies the model complexity but also dilute the communication pressure for requesting data. This evolutionary operation requires the model assess the importance of particular neighbor data to the prediction of its own data.

In this section, a evolutionary operation scheme called the *evolution framework* is introduced. It carries out evolutionary operations (some were mentioned above) to modify the embedded structure of a sensor node, with the aim to adapt to the environment and meanwhile reduce energy consumption. The *evolution framework* is guided by the vector $\mathcal{P}$, the so-called *evolution-operation-selector*, which is a probability vector for all possible operations (ref. *section-5.2.1*). The *evolution framework* starts from adjusting the value of $\mathcal{P}$ based on the fitness value of structure as well as constraints imposed by the applications, and decides which operation will be operated based on the value of $\mathcal{P}$. A sketch of such *evolution framework* is given in *fig-5.4*. In the following, details of the *evolution framework* will be discussed, with the order starting from $\mathcal{P}$ as shown in *fig-5.4*.

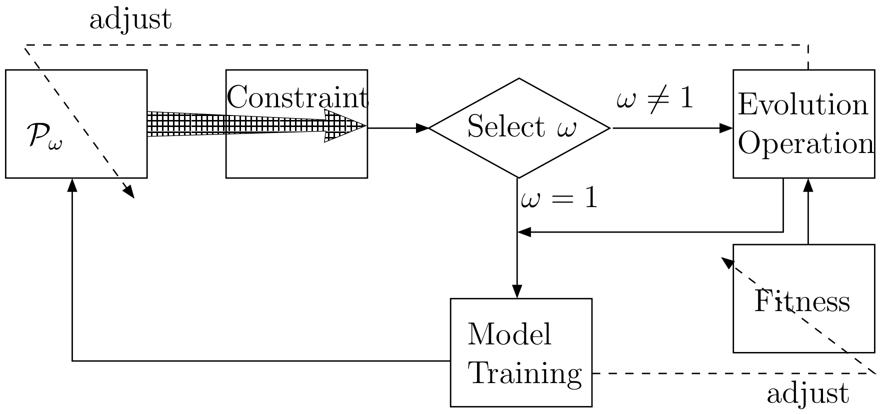


Figure 5.4: *Sketch of the* evolution framework

5.2.1 Evolution-Operation-Selector

Whether or not to conduct evolutionary operation and which evolutionary operation is going to be implemented are decided by vector $\mathcal{P}$, the so-called *evolution-operation-selector*. Vector $\mathcal{P}$ is a probability vector with each entry of $\mathcal{P}$ describes the probability of one particular operation being performed and it always satisfies *eq-(5.1)*.

$$\sum_{\omega} \mathcal{P}_\omega = 1 \tag{5.1}$$

The first entry of $\mathcal{P}$, *i.e.* $\mathcal{P}_1$, represents the probability of "null operation", that is, nothing concerning evolution will be done but normal on-line training of weights in neural network is performed. The value of $\mathcal{P}_1$ will be initialized with high value, such that no evolutionary operations will be conducted at the beginning before the neural network is considered to be mature, *i.e.* the model is somehow well trained in terms of the current structure. Rest of the entries in $\mathcal{P}$ are those probabilities of evolutionary operations (refer to *section-5.2.5*), with small values at the beginning. Each time step, values of probability for evolutionary operations will increase by small amount as in *eq-(5.2)*, so that the probability of them being chosen is getting higher along time. Note that φ denotes a small fluctuation, and a typical setting is $\varphi = 10^{-3}$.

$$\mathcal{P}_\omega = \mathcal{P}_\omega * (1 + \varphi) \ , \quad \omega \in [2, \dots] \tag{5.2}$$

Addition to the regular increment as in *eq-(5.2)*, value of respective entry of particular evolutionary operation in $\mathcal{P}$ may also be altered based on the constraints (ref. *section-5.2.2*) imposed by the application.

Index of active operation is first randomly chosen according to the probability represented by $\mathcal{P}$ value. With the selected index, the respective operation will be further considered in terms of the constraints of application (ref. *section-5.2.2*). Once such operation is not contradictory to the constraints it will be implemented afterward and the respective value in $\mathcal{P}$ is cut down and have to wait for the next time when it is picked; if the operation is contradictory to the constraints it will be skipped and the "null operation" will be implemented instead.

5.2.2 Constraint Conditions

Constraint conditions are based on the consideration of energy consumption, memory capacity limits, the sensor nodes' processing capabilities and other constraints related to the particular application. Depending on applications, one can specify other constraint. These constraint conditions set up boundaries for model properties, in such a way that a candidate operation (as mentioned in *section-5.2.1*) is rejected if the proceeding of such operation will lead to the breach of boundaries.

In the following, those constraints considered in simulation in *chapter-6* will be addressed:

(1) processing complexity ($\mathscr{C}_s$). Size of the embedded learning model (number of trainable parameters in the neural network model) determines the complexity of the learning model. The limit in processing capacity of the embedded processor of sensor node hinders the learning model from employing the model structure of large network size.

(2) maximum and minimum number of input data streams (n_{xds} and n_{nds}). Number of input data streams, *i.e.* the dimension of the input data, can affect the energy consumption because of the communication requirement. A smaller value of this number

implies that the sensor can reduce its effort of listening to neighbors for the input data. But on the other hand, rich information on the environment, which is realized by increasing number of input whose data is transferred from the neighboring sensors, helps the sensor to build up a precise prediction model and thus enhance effect of the *duty-cycle reduction*. Therefore, there is always the compromise between the prediction model accuracy and the energy consumption, which leads to the contradiction between values of n_{xds} and n_{nds}. And the conflict between them is also depended on the application details.

(3) maximum and minimum redundancy of data (n_{xrd} and n_{nrd}). In order to remedy the situation that one particular sensor is temporarily inaccessible, redundancy of the respective data in the sensor network is very helpful. Such redundancy exists in the form that another sensor node is using its value for the prediction and at the same time will have a predicted value for the inaccessible sensor. When the prediction at the spot where inaccessible sensor locates is required, the redundancy of the measurement can serve as an alternative. Large value in n_{xrd} could lead to large complexity of embedded neural network, while small amount of n_{nrd} could make the sensor network less robust.

(4) recover time (t_{rc}). Right after the evolutionary operation, the structure of the *SpiralRNN* model is changed abruptly and prediction performance of the model can be unacceptable. To ensure the novelty of new *SpiralRNN* structure is preserved, the value of t_{rc} should be large enough such that the *SpiralRNN* model is prevented from evolution before the model becomes mature.

5.2.3 Model Training

The principle of training of the embedded learning model is similar to the one described in *section-3.2.2*. The difference lies in that, when predicted data instead of transferred data are used for training, the residual value between output and target of those respectively replaced data is set to zero, such that the adjustment of parameters in the *SpiralRNN* model is only subject to the residual of its own measurement data. Let m be the index of sensor's own measurement data in the input/output vector, and $T^\dagger$ be set of time steps when transferred data is replaced by predicted data, the residual value of i^{th} output at time step t is calculated as followed:

$$\delta_{i,t} = \begin{cases} 0 & t \in T^\dagger,\ i \neq m \\ \hat{y}_i - y_i & t \in T^\dagger,\ i = m \\ \hat{y}_i - y_i & t \notin T^\dagger \end{cases} \tag{5.3}$$

As it will be mentioned later, the evolutionary operations alter the structure of *SpiralRNN*, and change the combination of connection weights. The covariance matrix P in extended Kalman filter (ref. *eq-(3.33)*) is modified according to the modification of the connection weights combination. In particular, when structure components are pruned, the connection weights will be removed from the training set, and matrix P is shrunken, in such a way that the corresponding rows and columns in matrix P that related to the deduced connection weights will be removed (ref. *fig-5.5(a)*). When a new component is added, the training set and the matrix P are extended in block-matrix manner (ref. *fig-5.5(b)*), where all the new entries are zeros except the new diagonal entries are set to typical values, *e.g.* "1".

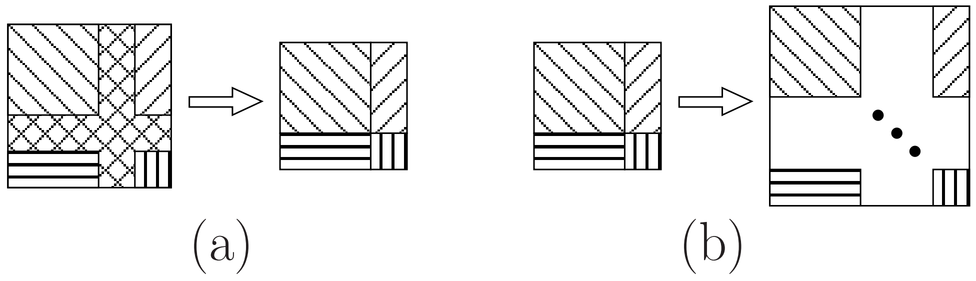


Figure 5.5: *Modification of matrix P in extended Kalman filter when structure of* SpiralRNN *is changed. (a) Matrix P is shrunk by removing the corresponding rows and columns; (b) Matrix P is enlarged by extending the diagonal and filling with values equal "1", where the new diagonal entries are represented by dots and the empty areas are entries equal "0".*

5.2.4 Fitness Values

Three types of fitness values are required in this context, including the fitness value for the learning model as a whole, the fitness value for each hidden units in the *SpiralRNN* model and the fitness value for each data stream of the sensor node in question. All of these fitness values indicate how much does the respective structure fit the current situation, where the larger the value is the better the respective structure fits. These fitness values are introduced in following:

1. Fitness value for sensor node as a whole indicates how well the learning model in the sensor fits with the circumstance. This fitness value is useful to find out the best neighbor adapting to the circumstance. Based on fitness values from one sensor's neighbors, the sensor can be aware of the strength of its neighbors which help to make further decision. Calculation of this fitness value is based on the current fitness $\tilde{f}_c$ at each time step in *eq-(5.4)*, then a Kalman filter is applied to calculate its mean value. *Eq-(5.4)* takes into account both the performance of sensor node together and the model complexity. As these two factors can not be compared directly, a multiplication operation between them is superior to a weighted summation operation as usual, but

the coefficient α should be carefully selected based on the requirement of application.

$$\begin{aligned}\tilde{f}_c &= -\left(\frac{n_{wei}}{N}\right)^{\alpha} E^* && (5.4)\\ f_t &= \mathbf{Kf}\left(\tilde{f}_c\right) && (5.5)\\ \text{where } E^*_i &= \frac{|\epsilon_i|}{\sigma_i} \quad i \in [1,\ldots,d]\end{aligned}$$

where $\tilde{f}_c$ is the current estimation, function $\mathbf{Kf}$ stands for the Kalman filter which estimates the mean value, $\epsilon_i = \hat{y}_i - y_i$ is the direct output error of the *SpiralRNN* model, n_{wei} and N are respectively the actual number of weights in the *SpiralRNN* model and the maximum number of weights allowed, α is the coefficient associated with the weight-ratio, σ is the standard deviation of data which is also online-estimated by a Kalman filter, i is the index in I/O list of the learning model. The calculation of the standard deviation σ of variable X is based on the definition $\sigma^2 = \overline{\left(X - \overline{X}\right)^2}$, which can be realized by:

$$\sigma^2 = \mathbf{Kf}\big(\left(\hat{y}_i - \mathbf{Kf}\left(\hat{y}_i\right)\right)^2\big). \qquad (5.6)$$

2. Fitness value for spiral unit serves for the purpose of implementing evolution based on ranks of all the spiral units inside one sensor node, where the ranking is the determined by the respective fitness value. In particular, it is prerequisite for *No.*(1) and *No.*(3) evolutionary operations in *section-5.2.5*. As shown in *eq-(5.7)*, fitness value is computed as the sum of the absolute gradient value of output *w.r.t.* associated hidden state vector of the k-th spiral unit, which indicates the importance of corresponding hidden-state vector.

$$\begin{aligned}\tilde{f}^{(k)}_c &= \sum \left|\frac{\partial y_t}{\partial s^{(k)}_t}\right| \quad k \in [1,\ldots,n_{_{units}}] && (5.7)\\ f^{(k)}_t &= \mathbf{Kf}\left(\tilde{f}^{(k)}_c\right) && (5.8)\end{aligned}$$

In the equation shown above, variable $\tilde{f}^{(k)}_c$ stands for the current estimation of fitness, function $\mathbf{Kf}$ again refers to the Kalman filter operation, y_t represents the output vector of *SpiralRNN* model at time step t, $s^{(k)}_t$ is the hidden state vector in k-th spiral unit in question at time t, notation k and $n_{_{units}}$ follow *chapter-3* and respectively refer to the index for hidden unit and the total number of spiral units in the *SpiralRNN* model.

3. Data stream also has its fitness value which describes the dependence of the *SpiralRNN* model of one sensor node on the respective data stream. One natural way is to measure the covariance between different data streams, which indicates how relevant is the neighbor data stream related to the measured data stream from sensor itself. Covariance of two variables X and Y can be calculated according to *eq-(5.9)*, where

$\overline{X}$ refers to the mean value of variable X.

$$COV(X,Y) = \overline{\left(X - \overline{X}\right)\left(Y - \overline{Y}\right)} \tag{5.9}$$

Therefore, the fitness value of data stream $\hat{y}_t^{(i)}$ is equal to the mean value of

$$(\hat{y}_t^{(i)} - \bar{y}_t^{(i)})(\hat{y}_t^{(m)} - \bar{y}_t^{(m)}), \quad \forall i \in [1,d] \text{ and } i \neq m,$$

where i is the index for the i-th data stream of the sensor in question, m $(1 \le m \le d)$ is the particular index of the data stream measured by sensor itself and $\bar{y}_t^{(i)}$ is the mean value of data stream $\hat{y}_t^{(i)}$. Calculation starts with the temporal variable $f_i^\dagger$ for each index i, where the normalization is taken in order to avoid the misleading when value of data stream relatively remains constant and calculation of σ is according to *eq-(5.6)*.

$$f_i^\dagger = \frac{\hat{y}_t^{(i)} - \mathbf{Kf}\left(y_t^{(i)}\right)}{\sigma_i}, \quad \forall i \in [1,d]$$

As two variables are regarded as irrelevant when the absolute value of covariance is small, the current estimation of fitness is given as $\tilde{f}_c^{(i)}$ in *eq-(5.10)* with absolute operation and the fitness value takes the mean estimation by Kalman filter.

$$\tilde{f}_c^{(i)} = \left|f_i^\dagger f_m^\dagger\right| \qquad i \neq m \tag{5.10}$$

$$f_t^{(i)} = \mathbf{Kf}\left(\tilde{f}_c^{(i)}\right) \quad i \neq m \tag{5.11}$$

As mentioned, in the calculation of fitness values, Kalman filter (**Kf**) is heavily used in estimating the corresponding mean values of different variables. Since the subject of estimation in this case is the same subject of measurement and there is no dynamics evolution for the estimated variable, the update-and-measurement steps of Kalman filter in *eq-(2.26)* and *eq-(2.27)* are simplified as:

$$x_t = x_{t-1} + \mu_t \tag{5.12}$$

$$z_t = x_t + \nu_t \tag{5.13}$$

Looking for a stable result of estimation, the settings $Q_t = 10^{-8}$ and $R_t \sim \mathscr{N}(0,1)$ are taken and are fixed after initialized, and P_t is initialized with a random value taken from the normal distribution $\mathscr{N}(0,1)$. For the relationship between variables P_t, Q_t and R_t, more details can be found in *section-2.2.3*.

5.2.5 Evolutionary Operations

Within each sensor node, evolutionary operations will take a particular hidden unit or input nodes for data streams as genes for evolutionary operations. Implementation of these evolution operations, satisfying those constraints mentioned in *section-5.2.2*, modifies the structure of *SpiralRNN* in sensor nodes. The first three operations are related to the spiral unit and the last two operations concern on the data stream of *SpiralRNN*:

(1) to remove a hidden unit. The choice of the spiral unit to be removed is determined by the fitness value $f_t^{(k)}$ in *eq-(5.7)* of spiral units. The spiral unit with the least value in $f_t^{(k)}$ will be chosen and be removed from the structure. Certainly, this operation will be bypassed when the embedded *SpiralRNN* model contains only one spiral unit.

(2) to create and insert a new hidden unit. A new hidden unit can be created and inserted under the condition that such insertion will not cause the model complexity $\mathscr{C}_s$ exceeding the constraint limit. The structure of the new hidden unit, following the description in *section-3.1.1*, is similar to that of other hidden units but with randomized amount of hidden nodes.

(3) to adopt a hidden unit from neighbor sensor. The procedure contains the following sequential steps (assuming the sensor in question is named sensor "A"):

   ① choose the neighbor sensor which has the best fitness value f_t (ref. *eq-(5.4)*) in the neighborhood, and name it sensor "B";

   ② from the sensor "B", choose the hidden unit with the best fitness value $f_t^{(k)}$ (ref. *eq-(5.7)*) under the condition of satisfying the complexity limit of sensor node;

   ③ merge the selected hidden unit from sensor "B" to sensor "A";

   The new hidden unit in sensor "A" remains changed values of its hidden connections and the connections from common input nodes [31] to newly added hidden nodes. The other connection values will be initialized satisfying with the distribution $\mathscr{N}(0, 0.01^2)$.

(4) to remove data stream. The data stream with smallest fitness value $f_t^{(i)}$ (ref. *eq-(5.10)*) will be removed from the model. This causes the connections from all hidden nodes to the corresponding input node and output node being erased. Note this only removes the corresponding data stream from the learning model, the physical communication connection can be resumed whenever it is required.

(5) to re-insert a data stream. Re-insertion of a data stream will re-introduce the selected data stream to the learning model. When there exist data streams being available for re-insertion (*i.e.* they were previously removed from the model, and current model does not depend on this data stream), the eligible data stream with the strongest $f_t^{(i)}$ (ref. *eq-(5.10)*) fitness value[32] will be selected and added as one of input data streams.

[31] Sensor "A" has different combination of data streams as sensor "B", whereas the intersection of the data streams between sensor "A" and "B" is corresponding to the common input nodes between them.

[32] Fitness value $f_t^{(i)}$ of removed data streams will be kept unchanged after the being removed from input layer.

Those corresponding new connections are initialized with values $\sim \mathcal{N}(0, 0.01^2)$.

Note that, in order to avoid too much change in the model structure, only one of mentioned operations will be conducted at one time. Furthermore, the recover time t_{rc} also prevents two evolutionary operations from being implemented within a short time.

5.2.6 *Evolution Framework* in a Nutshell

Combining the aforementioned issues, *table-5.2* has shown the *pseudo*-code for the proceeding of the *evolution framework* at one time step, which is corresponding to *fig-5.4*.

```
choose the operation according to the probability values in $\mathcal{P}$;
estimate the proposed values of respective properties
        when the chosen operation was performed;
if the proposed values against any constraint condition
    reject the chosen operation;
else
    perform the selected evolutionary operation with
            the aid from the fitness values;
    reduce the corresponding probability value in $\mathcal{P}$;
end
perform the normal on-line training operation;
update the fitness values;
update the probability values in $\mathcal{P}$;
```

Table 5.2: *Pseudo code for the* evolution framework.

Being able to modify the structure of embedded model whereas a longer time is required to recover from the structure modification, the *evolution framework* is useful at least in:

1. the applications where sensors measure the dynamical environment properties such that modification of the embedded model is allowed.
2. the applications with dense sensor network and many sensors measuring the same variable such that redundant data prevail and occupy the communication brandwidth.
3. the sensor network applications emphasizing the long-term performance and able to endure deficits in short-term.

Chapter 6

Simulations in Sensor Network Applications

In this chapter, the spiral recurrent neural network (*SpiralRNN*) model (ref. to *chapter-3*) together with the *duty-cycle reduction* and *evolution framework* (ref. to *chapter-5*) are applied in simulations, where the task for the sensor network is to predict temperature values.

In the following, settings for simulations are given first, and it is followed by the simulation results. At last, the discussion, focusing on the energy consumption and the maintenance of the prediction performance, is given.

For simplicity, the term “communication energy” is employed referring the energy spent in communication; similarly, the term “computation energy” refers the energy involved in data processing.

6.1 Simulation Settings

Assume there is a field with a regular grid of 11×11. Four heat sources and ten sensor nodes are located randomly inside the grid field, but they are not overlapped each other. Each sensor node has one temperature measurement unit, one communication module and one processing unit where the *SpiralRNN* model is employed as the embedded learning model.

Temperature values at positions of heat sources are dependent on the strength of heat source which obeys the dynamics shown in *eq-(6.1)* to *eq-(6.4)*. Because of radiation and thermal conduction, heat energy diffuses from heat sources to their surrounding, such that the temperature in the domain is varying in terms of time and location. Sensor nodes are required to train their learning models for temperature prediction, meanwhile mitigate the energy consumption.

Without explicit notification, one simulation consists of a training phase, which lasts for $5e3$ time steps, and a prediction test phase, which lasts for 50 time steps. For statistic and generality reasons, simulations will be implemented 30 times, where in each simulation the arrangement of the network (sensor node position, communication topology *etc.*) as well as that of the heat source (source position, value of temperature *etc.*) will be changed at random.

Heat source and temperature

Totally there are four heat sources in the field, where their positions are randomly chosen from the grid points. Each heat source emits certain amount of heat energy into the surrounding. The amount of emitted heat energy is variant at different time such that the strength of heat sources has similar dynamics as the *Lorenz* equations:

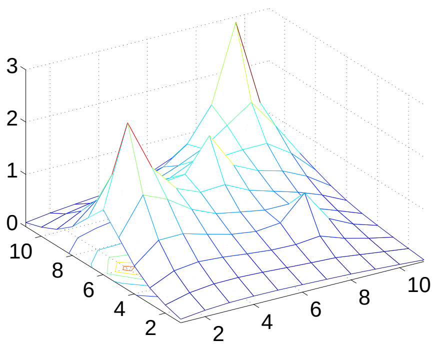
$$\begin{aligned} \dot{h}_1 &= 16(h_2 - h_1 - h_4) + h_3 && (6.1)\\ \dot{h}_2 &= (40 - h_3)h_1 - 0.5h_2h_4 && (6.2)\\ \dot{h}_3 &= h_1h_2 - 6h_3 && (6.3)\\ \dot{h}_4 &= 0.1h_3(h_1 - h_2) + 0.1h_3 && (6.4) \end{aligned}$$

where $\dot{h}$ refers to the time derivative of h, $\dot{h}_i$ refers to the i^{th} entry of $\dot{h}$ and each entry of vector $\vec{h}$ indicates the temperature change at particular heat source with $[0.1, 0.1, -0.1, -0.1]$ as the initial value of $\vec{h}$. Starting with value $\vec{T}_0 = 0$, temperatures at the heat sources can be calculated from *eq-(6.5)*, where μ is the noise satisfying $\mathscr{N}(0, 0.01^2)$. For convenient reason, values of temperature $\vec{T}$ are scaled down by factor of 10, such that $\vec{T}$ value is within a reasonable range. *Fig-6.1* illustrates the scenario data, where *fig-6.1(a)* and *fig-6.1(b)* respectively present the 3-D mesh plot and the contour plot of the temperature profile in the grid field at an instance, *fig-6.1(c)* shows the heat sources' temperatures which vary over time, and *fig-6.1(d)* gives examples of temperature over time that measured by sample sensor nodes.

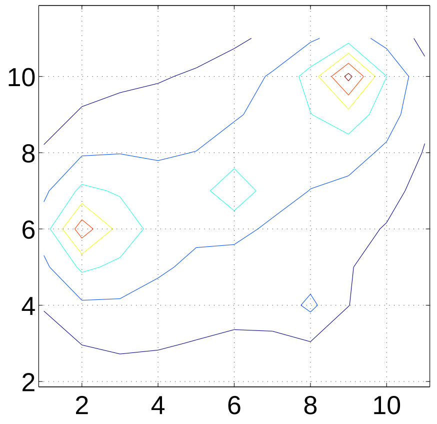
$$\vec{T}_{t+1} = \vec{T}_t + 0.01\vec{h} + \mu \qquad (6.5)$$

Diffusion

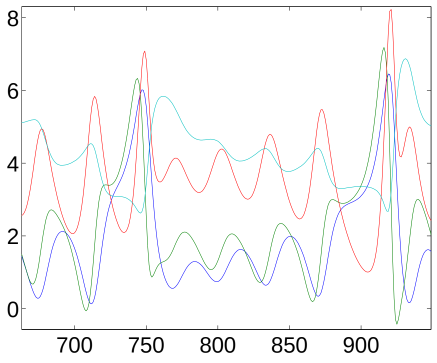
With heat energy diffusing through the domain, the temperature at any position in the space has variant values at different time. To calculate the temperature of these grid-points, let $u^t_{i,j}$ represents the temperature at grid point (i, j) at time t. It is known that, at one position, the difference of heat energy at different time is proportional to the difference between the amount of energy flowing in and the amount of energy flowing outwards. Assuming this is a homogeneous domain, the



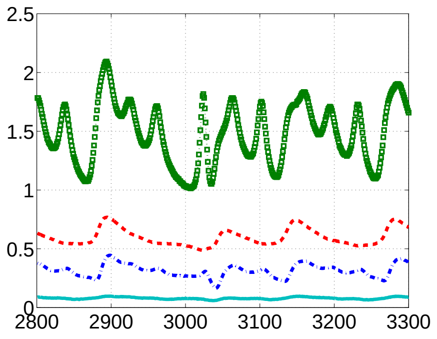
(a) mesh plot



(b) temperature contour



(c) heat sources' temperature



(d) measured temperature

Figure 6.1: *Data example of heat diffusion simulations. (a) A 3-D wire-frame mesh plot of the temperature in the grid. Note that the temperature surface is dynamically changing, and the plot only shows the temperature at one instance; (b) The corresponding contour plot of the temperature surface; (c) Temperature values of heat sources; (d) Measured temperature values from 4 sample sensor nodes.*

aforementioned relation is also applicable to the temperature. Therefore, *eq-(6.6)* holds.

$$\frac{u^{t+1}_{i,j} - u^{t}_{i,j}}{\Delta t} = k_d \frac{u^{t}_{i-1,j} + u^{t}_{i+1,j} + u^{t}_{i,j-1} + u^{t}_{i,j+1} - 4u^{t}_{i,j}}{\Delta x^2} \tag{6.6}$$

where constant k_d is the diffusion coefficient depending on the materials involved, Δt is the difference in time, Δx is the distance between two grid-points and is identical for any two grid-points. Solve the above equation, such that:

$$\begin{aligned} u^{t+1}_{i,j} &= \frac{k_d \Delta t}{\Delta x^2} \left(u^{t}_{i-1,j} + u^{t}_{i+1,j} + u^{t}_{i,j-1} + u^{t}_{i,j+1}\right) + \left(1 - 4\frac{k_d \Delta t}{\Delta x^2}\right) u^{t}_{i,j} \\ u^{0}_{i,j} &= 0 \end{aligned}$$

Note that, the temperature in the grid field is initialized as zeros. For simulation purpose, the following parametrization is taken:

$$s = \frac{k_d \Delta t}{\Delta x^2} = 0.15. \tag{6.7}$$

Let $\vec{U}$ be a vector listing temperature values of all grid-points one column after another, therefore $\vec{U} \in \mathbb{R}_{121\times 1}$ because of the grid size 11×11. The numerical solution of the heat diffusion is realized by *eq-(6.8)*, with the initial condition $\vec{U}_{t=0}$.

$$\vec{U}_{t+1} \;=\; sH\vec{U}_t + \nu \tag{6.8}$$

where $\nu \sim \mathcal{N}(0, 0.01^2)$ is Gaussian white noise, matrix H is a block-tridiagonal system with sub-block matrix A in the diagonal position and sub-block matrix $\mathbf{I_d}$ in the off-diagonal position. The matrix A is further a tridiagonal matrix with value "-4" in diagonal entries and value "1" in off-diagonal entries, as shown in *eq-(6.9)*. Matrix $\mathbf{I_d}$ is the identity matrix which has the same size as matrix A.

$$H = \begin{pmatrix} A & \mathbf{I_d} & & \\ \mathbf{I_d} & \ddots & \ddots & \\ & \ddots & \ddots & \mathbf{I_d} \\ & & \mathbf{I_d} & A \end{pmatrix}_{121\times 121} \qquad A = \begin{pmatrix} -4 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -4 \end{pmatrix}_{11\times 11} \tag{6.9}$$

Note that $\vec{U}_t$ in *eq-(6.8)* represents the temperature vector for all grid points in the domain, including those grid points for heat sources and sensor nodes. As it is mentioned, temperature values at heat sources are solely determined by heat sources but not by diffusion. *Eq-(6.10)* is therefore applied.

$$\vec{U}_{t+1,\Lambda} \;=\; \vec{T}_{t+1} \tag{6.10}$$

where Λ is the set of indices in $\vec{U}_t$ for those grid points where heat sources locate. Entries in $\vec{U}_t$ corresponding to the sensor nodes position are the measured data of sensor nodes, and are taken as training data for the learning model.

Sensor Net

Ten sensor nodes are randomly spread across the grid-field, and constitute a static connected non-directed network via wireless connections. Many researches [89, 90] have proposed methods for the automatic construction of ad hoc networks. For the purpose of simulation, a simple semi-self-organized version is employed here, and it contains the following steps:

① randomly distribute ten sensor nodes in the grid field;

② gradually increase the communication range of each sensor node;

③ once sensor A has found sensor B within its communication range, it will take sensor B as its neighbor sensor. The directed connection "A→B" is then established.

④ each sensor continues the process No.3 until its neighbor number has reached "3"

⑤ convert all these directed connections recognized in step No.3 to un-directed mappings, such that once sensor node A recognizes node B as its neighbor, node B will also take A as neighbor.

⑥ manually check whether the network as a whole is connected. If not, repeat from the first step.

With this approach, one can obtain a reasonable connected network. *Fig-6.2* gives an example. Depending on the distance between the sensor node with the heat sources and also on the strength of the nearby heat source, the variation of temperature values through time is different. It is shown in *fig-6.1(d)*, as an example, that the temperature changes frequently at some location while the temperature at other locations is basically constant.

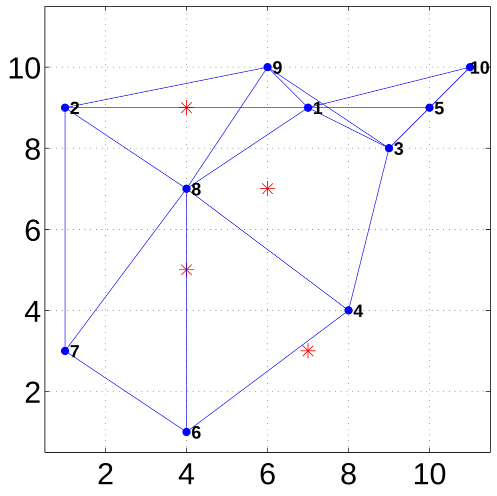
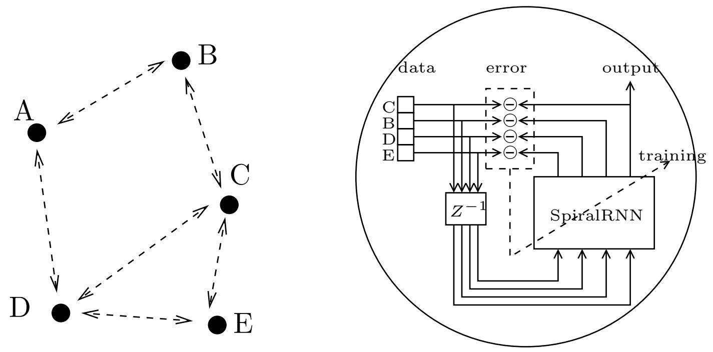


Figure 6.2: *Sample topology of sensor network. Red stars shows where the heat sources locate; blue dots represent ten sensor nodes; the number aside sensor node shows the ID of respective sensor node; lines between two sensor nodes refer to established wireless connections between them.*

Prediction model

Within each sensor node, a *SpiralRNN* model is deployed for the purpose of signal processing as *SpiralRNN* models are proven to be efficient and stable for on-line training. The number of input neurons of the *SpiralRNN* model is initialized as the number of sensor nodes in its neighborhood (including the sensor itself), such that the measurement data from each sensor node within the neighborhood is presented in the input layer of the *SpiralRNN* model, and apparently in the output layer as well. Note that, at any time instance, the number of input nodes must be identical to the number of output nodes, but the input number or output number can vary during the simulation because of the evolution. *Fig-6.3* gives an example of such a structure.

Parameters of the *SpiralRNN* model is configured mainly under the same way as described in *section-4.1*. The difference lies in that the number of hidden units and number of hidden nodes in each hidden unit will be randomly initialized, subject to the condition that the model complexity $\mathscr{C}_s$ (number of model parameters; ref. *page-45*) should not exceed 80% of the maximum allowed complexity value. This maximum allowed complexity value depends on the particular requirements of application, and is set to 100 or 200 in corresponding simulations.



(a) sensor net (b) SpiralRNN in node-C

Figure 6.3: *(a) An example of sensor network with 5 sensors, where sensor node "C" is connected to sensor nodes "B" "D" "E"; (b) The* SpiralRNN *model inside sensor node "C". The* SpiralRNN *model is trained with data from all 4 sensors inside the neighborhood of sensor "C". In the autonomous-test phase, only the output corresponding to its own measure data is considered.*

Settings for evolution

Referring to *section-5.2*, the *evolution framework* is guided by the value of $\mathcal{P}$. Entries of $\mathcal{P}$ represent the occurrence probabilities of corresponding operations. The initial value of vector $\mathcal{P}$ is $[0.8, 0.04, 0.04, 0.04, 0.04, 0.04]$, with the first entry corresponding to normal on-line training and the rest five entries corresponding to evolutionary operations (refer to *section-5.2.5*). In each training step, entry values for evolutionary operations in $\mathcal{P}$ slightly increase by 10^{-3}:

$$\mathcal{P}_i = \mathcal{P}_i(1 + 10^{-3}), \quad \forall i \in [2, 6], \tag{6.11}$$

in order to gradually increase the possibility of evolutionary operations. Choice of operations at each training step is based on the vector $\mathcal{P}$, a random value r_d

(ref. *section-5.2.1*) and application constraints (ref. *section-5.2.2*). Once an evolutionary operation is selected and is implemented, its respective probability decreases by 20%, in order to avoid successive repetition of respective evolutionary operation.

Parameters of application constraints (ref. *section-5.2.2*) are defined as following: values of n_{xds} and n_{xrd} are set to 4 in order to reduce the effort for computation and communication; values of n_{nds} and n_{nrd} are set to 2 in order to keep the efficiency of the embedded model and maintain the redundancy of measurement data of each senor node; for protection of novel structure, the recover time is set to $t_{rc} = 200$. The maximum complexity of embedded model in sensor node is set to $\mathscr{C}_s = 100$ and $\mathscr{C}_s = 200$ in different simulations, as will be mentioned later.

Training

No matter which operation is chosen based on the $\mathcal{P}$ value, the on-line training of model parameters is conducted, where parameter values are updated at each time step (Details of implementation of on-line training of the *SpiralRNN* can be found in *section-3.2.2*). Keeping the on-line training at every time step, the autonomous test starts at time step $t = 5001$, and ends at $t = 5050$. In each autonomous test, prediction is conducted for up to 15 steps, *i.e.* the model evolves itself 15 times (Details of autonomous test can be found in *section-3.2.3*).

Evaluation

Simulation evaluation focuses on the energy consumption as well as the prediction performance. Proposed criteria include: (1) logarithmic error evaluation on prediction (2) the amount of arithmetical operations n_p, the amount of transfer data n_m and their respective variance; (3) the amount of consumed electric charge based on the specs of TelosB sensor node.

(1) Performance evaluation of one sensor node focuses only on the prediction error of its own measurement data. For example, let $x_t^{(\tau)}$ denotes the τ-step ahead prediction at time step t, and $\hat{x}_t^{(\tau)}$ the corresponding target. Note that both of them are vectors. Without loss of generality, let m be the index of its measurement data in the input, then the evaluation is based on the particular residual of m-th entries: $\hat{x}_t^{(\tau)}(m) - x_t^{(\tau)}(m)$. Given the value of standard deviation σ of sensor's measurement data, the τ-step ahead prediction error at time t is given by *eq-(6.12)*. A mean value of $\varepsilon_t^{(\tau)}$ is calculated for the results of 50 successive time steps from $t = 5001$ to $t = 5050$, as shown in *eq-(6.13)*. The final prediction error of the entire sensor network is the respective average value of all sensors.

$$\varepsilon_t^{(\tau)} = \log_{10}\left(\frac{\left(x_t^{(\tau)}(m) - \hat{x}_t^{(\tau)}(m)\right)^2}{\sigma^2}\right), \quad \tau = 1, 2, \cdots \tag{6.12}$$

$$\varepsilon^{(\tau)} = \overline{\varepsilon_t^{(\tau)}}, \quad t \in [5001, \dots, 5050] \tag{6.13}$$

(2) The energy consumption is split into two parts: communication energy consumption and computation energy consumption. In real-world applications, the actual amount of consumed energy is related to the way how data processing and communication are handled. For example, as a shared medium network, sensor nodes transfer data according to channel access methods such as time division multiple access (TDMA). Generally speaking, a time frame under TDMA is divided into time slots of even interval, where each time slot is occupied by one of the sensor nodes to transfer data meanwhile other neighbor sensor nodes are switched to listening mode. The energy consumption is therefore related to the length of each time slot and the amount of time slots during which the sensor radio is activated. However, for the sake of simplicity, evaluation of energy consumption in the simulation is based on the amount of arithmetic operations and the amount of communicate data, as stated in the following. Note that all these values are calculated in the training phase of one run (*i.e.* one simulation that takes 5000 time steps)

- (n_m) The total amount of communication data of one sensor node during the training in one simulation. The value of n_m considers both of the data being sent out and received, indicating the amount of energy spent in communication.
- (n_p) The total amount of arithmetic operations in training for one sensor node in one simulation.
- (v_m) The variance of sensors' n_m values in one simulation, indicating the imbalance of communication energy consumption in the sensor network;
- (v_p) The variance of sensors' n_p values in one simulation, indicating the imbalance of computational energy consumption in the sensor network.

The values of n_m and n_p focus on the total energy of one sensor which have been consumed during the simulation, whilst the evaluation of v_m and v_p indicate the early depletion of sensor network in the case where one or couple of sensor nodes have run out of power much earlier than the others.

(3) With the estimated values of n_m and n_p, one can roughly estimate the energy which has been used in the simulations. Recalling the specification of TelosB sensor mote in *chapter-2*, the working current of device is 1.8mA when the 16-bit micro-controller MSP430 works on summit frequency of 8MHz. The amount of electric charge spent on n_p amount of arithmetic operation can be calculated as in *eq-(6.14)*, where the unit "mAh" stands for milli-Amp hour, and 3600 reflects that one hour has 3600 seconds.

$$\frac{n_p}{8 \times 10^6} \times \frac{1}{3600} \times 1.8\,(\text{mA}) = 6.25 \times 10^{-11} n_p\,(\text{mAh}) \tag{6.14}$$

With the activation of radio, working current of sensor rises to 21.8mA for receiving data and 19.5 for data transfer. For the sake of simplicity, it is assumed that the working current value equals 20mA in both receiving and sending operations. The radio bandwidth of TelosB is up to 250kbps. Each packet in communication between sensors should contain at least three entries - the measurement data itself, the ID of the sensor from which the data are sent and the time step when the measurement was made. Therefore, omitting overheads, communication of one piece of measurement data is actually required to transfer at least three piece of data. One can use *eq-(6.15)* to calculate the electric charge spent on data communication.

$$3 \times 16 \times \frac{n_m}{250 \times 10^3} \times \frac{1}{3600} \times 20\,(\text{mA}) = 1.067 \times 10^{-6} n_m\,(\text{mAh}) \qquad (6.15)$$

where value "16" is corresponding to the 16-bit operation of TelosB mote.

Eq-(6.14) and *eq-(6.15)* will be employed in calculation of energy spending in the discussion about early depletion in *section-6.3.3*.

Simulation in a nutshell

Summarizing all the mentioned concepts, the simulation is implemented with the following steps:

① Construct a sensor network with sensor nodes and communication connections;

② Prepare the temperature profile of heat sources with aid from *eq-(6.1)* to *eq-(6.5)*;

③ Complete the temperature profile of the entire domain by *eq-(6.8)* and *eq-(6.10)*;

④ From $\vec{U}$, extract the corresponding entries as sensor nodes' measurement data, which will be the training data of embedded learning models;

⑤ Initialize the embedded *SpiralRNN* models of all sensor nodes;

⑥ Implement the on-line training of the *SpiralRNN* model with measurement data for $5e3$ time steps; The *duty-cycle reduction* and *evolution framework* are applied;

⑦ Conduct data prediction for 50 time steps.

⑧ Assess the prediction performance and the energy consumption.

6.2 Simulation Results

Simulations are conducted under the guidance given in *chapter-5* and the settings of simulation in *section-6.1*. In the first comparison, the effects of the *evolution framework* (EF) and *duty-cycle reduction* (DCi) are emphasized by comparing with the so-called "normal" scheme. In the second demonstration, the impact of different activity period $\mathscr{A}_p$ values (ref. *section-5.1*) is addressed.

Note that the sensor network setting (including the position and topology of sensor nodes, and the temperature profile in the space) is different in each of 30 runs, and, in each run, the same setting is applied to different competing schemes. The maximum model complexity $\mathscr{C}_s$ determines the size of the model when it is initialized, whereas the actual value of model complexity is varying in schemes with the *evolution framework*.

6.2.1 Simulation 1

As the first experiment, several operation schemes are compared. They include:

① An "EF" scheme refers to the scheme using the *evolution framework* to modify the *SpiralRNN* structure;

② In a "DC2" scheme, the *duty-cycle reduction* is implemented with the activity period value $\mathscr{A}_p = 2$;

③ An "EF+DC2" scheme combines "EF" and "DC2" schemes together;

④ In the "normal" scheme, both *evolution framework* and *duty-cycle reduction* are not applied but only the normal on-line training of model parameters is implemented.

Table-6.1 has listed the prediction performance of these schemes, where the maximum complexity of the *SpiralRNN* is set to 100, *i.e.* $\mathscr{C}_s = 100$. The result has indicated that the "normal" scheme does outperform the other schemes in terms of prediction performance. The advantage of the "normal" scheme has been expected, because it enjoys the stability in structure and keeps the access to the measurement data of its neighbors all the time. But such advantage is not significant, if the standard deviation values of prediction errors are taken into account. *Fig-6.4* depicts the comparisons of aforementioned schemes in terms of 1^{st}-step to 15^{th}-step prediction. The extended bar coming from the curve of "normal" scheme refers to the standard deviation of prediction error in "normal" scheme. It has been shown that the difference between the "normal" scheme and other schemes is not significant, at least for short-term prediction.

Evaluating the energy consumption, *fig-6.5(a)(b)* illustrate the average amount of arithmetic operation per sensor node per simulation (n_p) and the average amount of communication data per sensor node per simulation (n_m); *fig-6.5(c)(d)* report their corresponding variance (the variance value among all sensor nodes in the network) values v_p and v_m.

As shown in *fig-6.5(a)(c)*, applying the *evolution framework* has helped the "EF" scheme and the "EF+DC2" scheme to gain the advantage in saving the computation energy, meanwhile the computation energy consumption among sensor nodes are more balanced, as with lower value in v_p. On the other hand, the communication energy can be reduced and balanced by employing the *duty-cycle reduction* as manifested in *fig-6.5(b)(d)*, where in the

	$\varepsilon^{(1)}$		$\varepsilon^{(5)}$		$\varepsilon^{(10)}$		$\varepsilon^{(15)}$	
	mean	std.	mean	std.	mean	std.	mean	std.
normal	-1.80	0.32	-1.40	0.27	-1.03	0.24	-0.78	0.24
EF	-1.74	0.30	-1.24	0.22	-0.84	0.19	-0.61	0.17
DC2	-1.75	0.28	-1.29	0.20	-0.90	0.19	-0.66	0.20
EF+DC2	-1.73	0.29	-1.19	0.22	-0.80	0.20	-0.58	0.20

Table 6.1: *Predictive error $\varepsilon^{(\tau)}$ ($\tau \in [1, 5, 10, 15]$) of different schemes ("normal", "EF", "DC2" and "EF+DC2" schemes). The maximum model complexity is set to $\mathscr{C}_s \simeq 100$. All $\varepsilon^{(\tau)}$ values are on a logarithmic scale.*

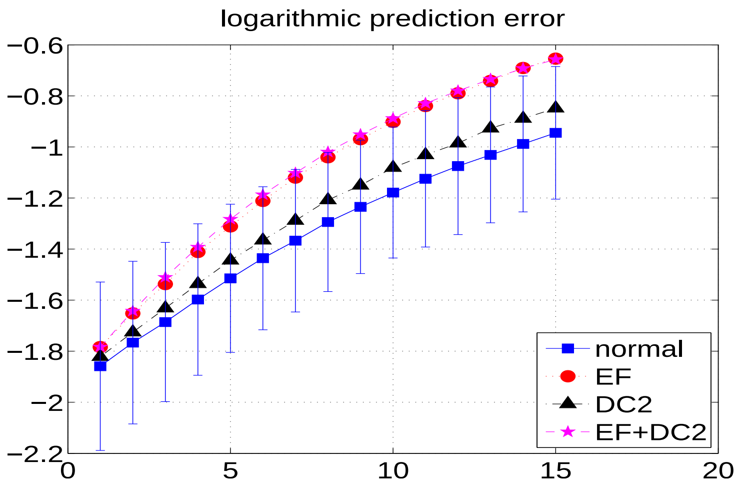
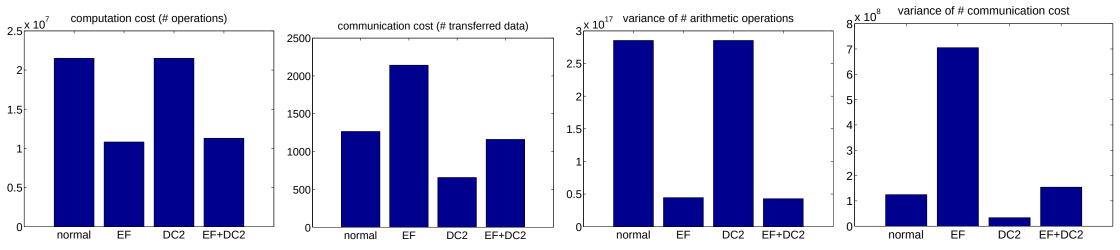


Figure 6.4: *Comparison in prediction error $\varepsilon^{(\tau)}$ ($\tau \in [1, \ldots, 15]$) of different schemes ("normal", "EF", "DC2" and "EF+DC2" schemes). The maximum model complexity is set to $\mathscr{C}_s = 100$. The X-axis stands for prediction step, and the Y-axis refers to the logarithmic prediction error value. Length of the extended bar represents the value of standard deviation of prediction error of the "normal" scheme.*

"DC2" scheme sensor nodes have saved 50% of energy in communication comparing with the "normal" scheme.

Increasing the maximum complexity $\mathscr{C}_s$ of the embedded *SpiralRNN* model to $\mathscr{C}_s = 200$, similar results are reported in *fig-6.6* and *table-6.2*, namely that the *evolution framework* and *duty-cycle reduction* respectively help to reduce the arithmetic operation effort and the communication effort.



(a) n_p (b) n_m (c) v_p (d) v_m

Figure 6.5: *Comparisons on the computation cost n_p (average amount of arithmetic operations per sensor node) and the communication cost n_m (average amount of communication data per sensor node) of different schemes, as well as their respective variance values. The maximum model complexity is $\mathscr{C}_s \simeq 100$. (a) average n_p value in the network; (b) average n_m value in the network; (c) variance v_p of n_p in the network; (d) variance v_m of n_m in the network.*

	$\varepsilon^{(1)}$		$\varepsilon^{(5)}$		$\varepsilon^{(10)}$		$\varepsilon^{(15)}$	
	mean	std.	mean	std.	mean	std.	mean	std.
normal	-1.85	0.32	-1.51	0.29	-1.17	0.25	-0.94	0.25
EF	-1.78	0.32	-1.31	0.26	-0.90	0.24	-0.65	0.22
DC2	-1.82	0.31	-1.44	0.25	-1.08	0.25	-0.84	0.24
EF+DC2	-1.78	0.33	-1.28	0.26	-0.88	0.25	-0.65	0.23

Table 6.2: *Predictive error $\varepsilon^{(\tau)}$ ($\tau \in [1, 5, 10, 15]$) of different schemes ("normal", "EF", "DC2" and "EF+DC2" schemes). The maximum model complexity is set to $\mathscr{C}_s = 200$ All values of $\varepsilon^{(\tau)}$ are on a logarithmic scale.*

6.2.2 Simulation 2

From the previous section, it is shown that the implementation of the *duty-cycle reduction* reduces the communication effort, and that implementation of the *evolution framework* scheme helps reducing the complexity of learning model. Combining both methods as "EF+DC2", one could obtain a better solution in terms of both n_m and n_p values. The comparison results with different sizes of the learning model are given in *fig-6.5* and *Table-6.1* for the $\mathscr{C}_s = 100$ case, in *fig-6.6* and *Table-6.2* for the $\mathscr{C}_s = 200$ case.

However, "EF+DC2" scheme didn't provide an outstanding advantage in communication energy consumption. This is so because the *evolution framework* inherently requires communication effort to obtain necessary information. On the other hand, the *evolution framework* is beneficial to minimizing the computational effort as well as to adapting the learning model to changing environment. It is therefore necessary, particularly in complicated applications which require intensive data processing. In order to obtain the advantage both

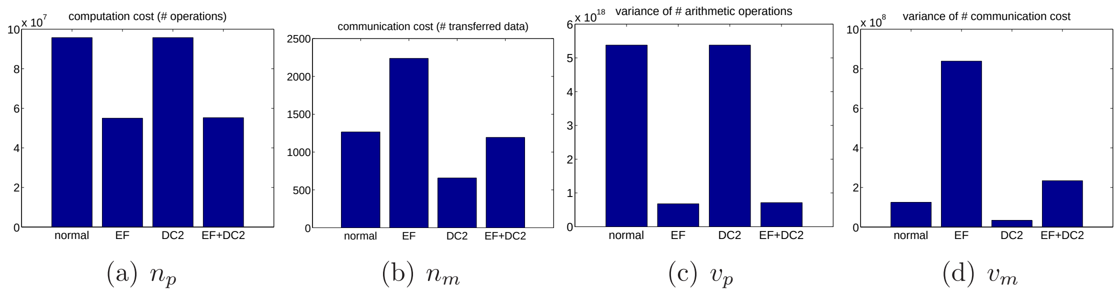


Figure 6.6: *Comparisons on the computation cost n_p (amount of arithmetic operations per sensor node) and the communication cost n_m (amount of communication data per sensor node) of different schemes and their respective variances. The maximum model complexity is $\mathscr{C}_s = 200$. (a) average n_p value in the network; (b) average n_m value in the network; (c) variance v_p of n_p in the network; (d) variance v_m of n_m in the network.*

in data processing and in communication, one straight-forward solution is to combine the *duty-cycle reduction* with the *evolution framework*, and meanwhile further increase the activity period value in *duty-cycle reduction* in order to achieve higher efficiency in energy spent on communication.

Let "EF+DCi" indicates the scheme with i being the value of activity period $\mathscr{A}_p$. For example, in "EF+DC3" scheme, sensor radio will be switched to the listening mode once in every three working cycles. Following simulations are conducted with "EF+DCi" schemes with different $\mathscr{A}_p$ values.

The results in *table-6.3* compares the average prediction performance $\varepsilon^{(\tau)}$ of sensors, where complexity of the embedded learning model is set to $\mathscr{C}_s = 100$. It is shown that the performance difference between different schemes in "EF+DCi" series is small in short-term prediction, even though the difference slightly expands when the prediction-step increases. While the increase in $\mathscr{A}_p$ value deteriorates the prediction performance to some extent, it helps to reduce the energy consumption in communication. This is manifested in *fig-6.7(b)(d)*. As expected, computational effort is reduced due to the application of the *evolution framework*, which is proven in *fig-6.7(a)(c)*.

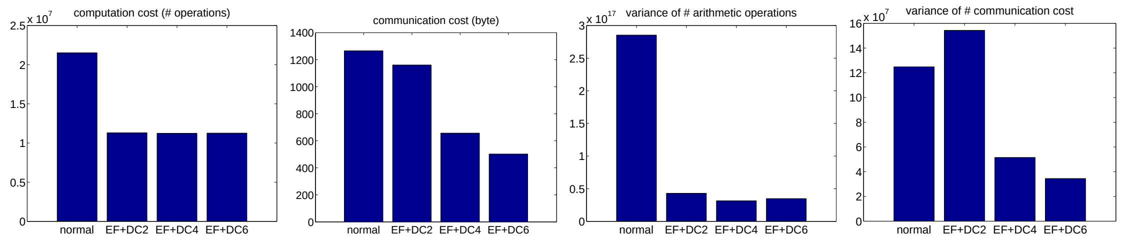
Simulation results corresponding to model complexity $\mathscr{C}_s = 200$ are given in *fig-6.8*, and *table-6.4*. They have all reported the similar result as discussed above.

6.3 Discussion

Based on the simulation results in previous section, some important issues in distributed sensor network applications will be addressed in the coming text. The discussion focuses

	$\varepsilon^{(1)}$		$\varepsilon^{(5)}$		$\varepsilon^{(10)}$		$\varepsilon^{(15)}$	
	mean	std.	mean	std.	mean	std.	mean	std.
EF+DC2	-1.73	0.29	-1.19	0.22	-0.80	0.20	-0.58	0.20
EF+DC3	-1.64	0.31	-1.06	0.24	-0.69	0.22	-0.48	0.22
EF+DC4	-1.61	0.26	-1.05	0.20	-0.67	0.19	-0.45	0.19
EF+DC5	-1.65	0.29	-1.05	0.19	-0.66	0.20	-0.45	0.21
EF+DC6	-1.64	0.30	-1.04	0.23	-0.64	0.22	-0.42	0.24

Table 6.3: *Comparisons of predictive error $\varepsilon^{(\tau)}$ ($\tau \in [1, 5, 10, 15]$) in "EF+DCi" series scheme. The maximum model complexity is set to $\mathscr{C}_s \simeq 100$ Note that values of $\varepsilon^{(\tau)}$ are on a logarithmic scale.*



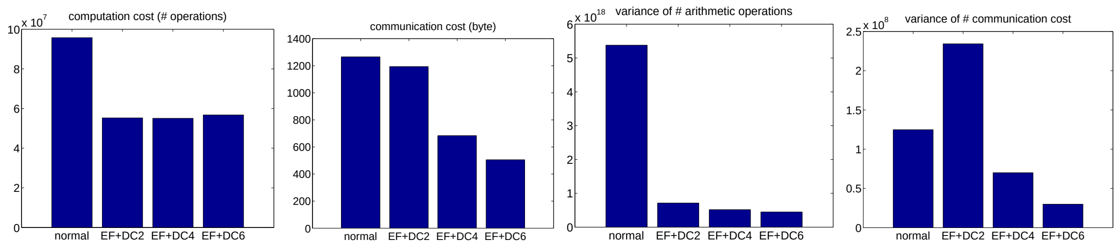
(a) n_p (b) n_m (c) v_p (d) v_m

Figure 6.7: *Comparisons of n_p and n_m between the "EF+DCi" series and the "normal" scheme and their respective variance. The maximum model complexity is set as $\mathscr{C}_s \simeq 100$. (a) Mean of sensors' n_p values in the network; (b) mean of sensors' n_m values in the network; (c) variance of different sensors' n_p values in the network; (d) variance of different sensors' n_m values in the network.*

	$\varepsilon^{(1)}$		$\varepsilon^{(5)}$		$\varepsilon^{(10)}$		$\varepsilon^{(15)}$	
	mean	std.	mean	std.	mean	std.	mean	std.
EF+DC2	-1.78	0.33	-1.28	0.26	-0.88	0.25	-0.65	0.23
EF+DC3	-1.73	0.30	-1.20	0.20	-0.81	0.17	-0.58	0.17
EF+DC4	-1.63	0.29	-1.10	0.25	-0.70	0.27	-0.47	0.27
EF+DC5	-1.66	0.28	-1.11	0.20	-0.69	0.22	-0.48	0.24
EF+DC6	-1.65	0.31	-1.10	0.23	-0.70	0.23	-0.48	0.24

Table 6.4: *Comparisons of predictive error $\varepsilon^{(\tau)}$ ($\tau \in [1, 5, 10, 15]$) in "EF+DCi" series scheme. The maximum model complexity is set to $\mathscr{C}_s \simeq 200$. Note that values of $\varepsilon^{(\tau)}$ are on a logarithmic scale.*

on the prediction performance of sensor model, the energy consumption in general, and the early depletion problem.



(a) n_p (b) n_m (c) v_p (d) v_m

Figure 6.8: *Comparisons of n_p and n_m between the "EF+DCi" series and the "normal" scheme and their respective variance. The maximum model complexity is set as $\mathscr{C}_s \simeq 200$ (a) Mean of sensors' n_p values in the network; (b) mean of sensors' n_m values in the network; (c) variance of different sensors' n_p values in the network; (d) variance of different sensors' n_m values in the network.*

6.3.1 Prediction Performance

Considering the prediction performance, the "normal" scheme without *evolution framework* and *duty-cycle reduction* has shown its strength by producing a lower prediction error ε. Depending on particular application requirement, such a difference is not significant, at least not for short-term prediction. This can be confirmed by the comparison of the normalized root mean square error (normalized RMSE) between the *DC2* scheme and the "normal" scheme on different prediction steps as shown in *fig-6.9*. The normalized RMSE is calculated by taking the square root value of the prediction error which is calculated similar to the value of $\varepsilon^{(\tau)}$ in *eq-(6.12)* and *eq-(6.13)* but without the "log_{10}" calculation. Positions of two curves in *fig-6.9* are slight shifted for sake of clearness. Each curve shows the respective mean value of the normalized RMSE as well as the respective maximum (upper bar) and minimum (lower bar) normalized RMSE values among the 30 simulations. It is shown, that the normalized RMSE values of these two schemes are very closed all the time, but there exists outliers in both schemes whereas the outliers have imposed more damage in longer-term prediction in *DC2* scheme because the lack of half of training data in the *DC2* scheme.

Comparably, "DC2" scheme has shown a better result than the "EF" scheme, particularly with the increase of complexity $\mathscr{C}_s$ of the embedded model (refer *table-6.1* and *table-6.2*). This is due to the fact that the "EF" scheme which will require a certain amount of time in order to recover.

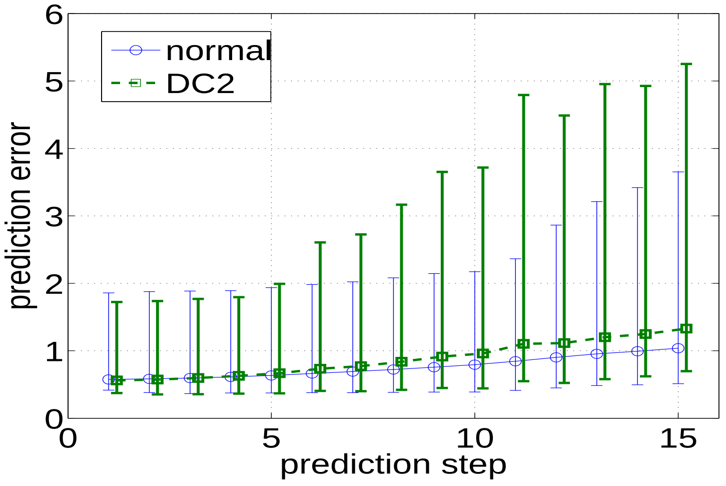


Figure 6.9: *Comparison of the normalized root mean square error between DC2 and the "normal" schemes on different prediction steps. Positions of two plots are slight shifted for sake of clearness. Each plot shows the mean normalized error as well as the maximum (upper bar) and minimum (lower bar) normalized error among the* 30 *simulations with one of the schemes.*

6.3.2 Energy Consumption

Results from the previous section have shown clearly that schemes using the *evolution framework* have an advantage in reducing arithmetic operations, and that schemes using *duty-cycle reduction* mitigate the effort for communication, though not for arithmetic operations.

Duty-cycle reduction

With the *duty-cycle reduction*, sensor nodes do not need to listen to their neighbors in every working cycle (even though they are still required to turn on the radio for sending out their own measurement data). Therefore, the number of communication time slots during when the radio is on has been reduced from $k+1$ to 1, where k is number of one sensor's neighbors. Thus, in the "DC2" scheme where sensor nodes need to listen to their neighbor once in two working cycles, the reduction rate r_k of energy spent on communication for sensor node with k neighbors is calculated in *eq-(6.16)* [87].

$$\begin{aligned} r_k &= \left(1 - \frac{2+k}{2(1+k)}\right) \times 100\% \qquad (6.16) \\ \lim_{k\to\infty} r_k &= 50\% \end{aligned}$$

Such calculation can be generalized for other "DCi" schemes as in *eq-(6.17)*, with i indicating the activity period $\mathscr{A}_p$ value. The limit value of $r_{i,k}$ implies that the communication-energy consumption in "DCi" scheme reduces to $\frac{1}{i}$ of the amount in the "normal" scheme.

$$\begin{aligned} r_{i,k} &= \left(1 - \frac{(i-1)+(1+k)}{i(1+k)}\right) \times 100\% \\ &= \left(1 - \frac{1}{i}\right)\left(1 - \frac{1}{1+k}\right) \times 100\% \qquad (6.17) \\ \lim_{k\to\infty} r_{i,k} &= \left(1 - \frac{1}{i}\right) \times 100\% \end{aligned}$$

Fig-6.10(a) illustrates the increase of the reduction rate with the rise in the neighbor number. For a sensor network with a connection degree[33] of 6, the *duty-cycle reduction* can save the communication effort by more than 40%. It is also observed that the gap between "DC2" and "DC3" is bigger than the difference between "DC3" and "DC4", and so on, which indicates the diminishing marginal utility of activity period $\mathscr{A}_p$ value. This can be explained by the variable f_{mu} defined in *eq-(6.18)*, which assesses the ratio of between the increment in reduction rate and the current reduction rate. The ratio f_{mu} of increment decreases when the activity period $\mathscr{A}_p$ increases. Note that the f_{mu} value is independent from the neighbor number k, and that i refers to the activity period $\mathscr{A}_p$ value.

$$f_{mu}(i) = \frac{r_{i,k} - r_{i-1,k}}{r_{i-1,k}} = \frac{1}{i(i-2)}, \quad i = 3, \cdots \qquad (6.18)$$

The distribution of f_{mu} given in *fig-6.10(b)* shows how much proportion of communication energy has been saved by increasing the value of activity period $\mathscr{A}_p$ by one. The value decreases from 33% at $i = 3$ to 12% at $i = 4$, and further falls down to less than 5% from $i = 6$.

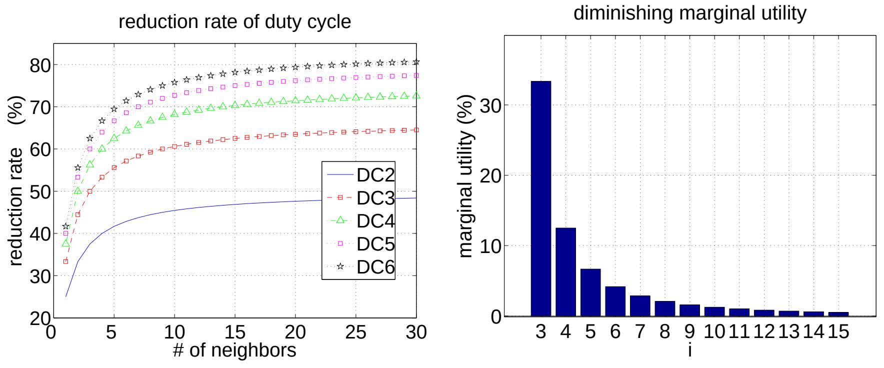
Even though the reduction rate is growing when $\mathscr{A}_p$ value increases, the gain from the increase of $\mathscr{A}_p$ value is getting smaller. Furthermore, the increase in $\mathscr{A}_p$ value implies that the embedded model has been trained by less genuine data than before, which will deteriorate the prediction performance. Therefore, one needs to compromise between a higher reduction rate in energy consumption and a better prediction performance. However, this is apparently depending on the particular application.

Evolution framework

Compared to the "normal" scheme, the *evolution framework* has saved the computation effort (n_p value) respectively by 50% (ref. *fig-6.5*) and by 40% (ref. *fig-6.6*).

On the other hand, the amounts of communication data (n_m value) of the "EF" scheme indicate that the scheme is more expensive in communication. However, the disadvantage of *evolution framework* has been exaggerated, because:

[33] The "magic number" to keep the connectivity of a radio network is six [91].



(a) reduction rate (b) marginal utility of $\mathscr{A}_p$ (i value)

Figure 6.10: *(a) Reduction rates of "DCi" schemes versus the number of neighbors. (b) The diminishing marginal utility of activity period $\mathscr{A}_p$.*

① The data required by evolutionary operations can be received together with measurement value from the same neighbor sensor.

② Particular in TDMA channel access method, the number of sensor's neighbors is the primary factor that determines the duration of time slots for receiving, which in fact affects the energy consumption.

Thus, the energy consumption for communication in the "EF" scheme has been overestimated, even though the amount of transferred data in the "EF" scheme is indeed higher than that of the "normal" scheme.

Furthermore, the occurrence of evolutionary operations decreases when *evolution framework* gradually alters the model structure. The energy consumption spent in transferring data, which is necessary for evolutionary operations, is therefore shrinking along with the occurrence frequency. *Fig-6.11* shows the convergence of model structure by illustrating the probability of evolution-operations occurrence per sensor node. The X-axis refers to the training time step, and the Y-axis stands for the average (over a time window of length 200 time steps) percentage of evolution-operations occurrence per sensor node in "EF+DC6" scheme. The decrease in probability manifests the convergence of embedded model of sensors. Note that the value of recover time (ref. *page-98*) is set to $t_{rc} = 200$, thus the upper limit of the average probability, that one sensor conducts evolutionary operation at one time step, is equal to 0.5%.

On the contrary, the *evolution framework* can be beneficial to reduction of communication by removing[34] irrelevant data streams. With the *evolution framework* changing the dependency of model, the amount of model inputs can be altered. *Fig-6.12*, shows the histogram (over 30 simulations) of the average amount of model inputs per sensor node after 5000 training step. The X-axis presents the average

[34] The removal of irrelevant data stream is motivated by the fact that some neighbor data is not important for the prediction of its own measurement data. See to *fig-6.1(d)*.

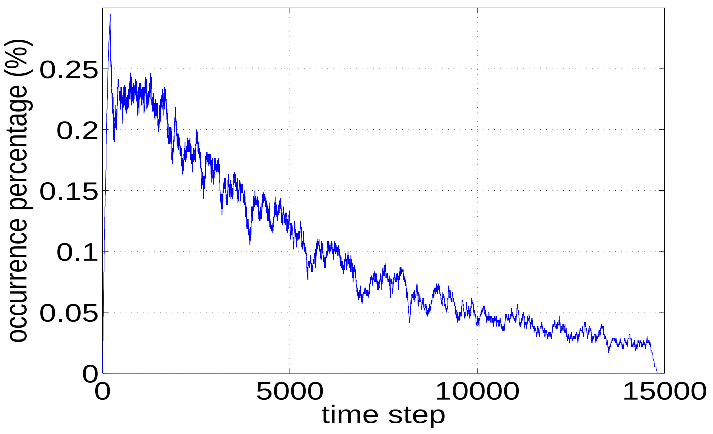


Figure 6.11: *The occurrence probability of evolution operations. The X-axis represents the training time step; the Y-axis refers to probability that evolutionary operation is conducted in sensor at one time step. Each value is calculated as the average within a time window of length 200.*

number of input neurons per sensor node, the Y-axis indicates the occurrence among 30 simulations. The higher amount of model inputs of sensor node is, the more communication effort is required to fetch measurement data. For the "normal" scheme which doesn't have evolution operation, the average amount of model inputs is about 4.8; the counterpart value for the other schemes with *evolution framework* is about 2.5. The ratio of communication activities is calculated as:

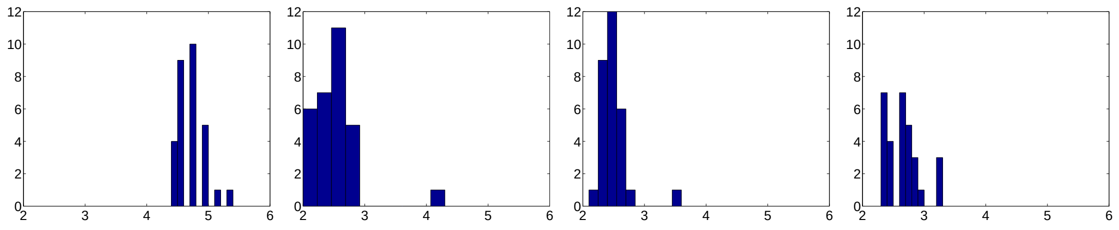
$$\frac{2.5-1}{4.8-1} \times 100\% = 39.47\%$$

This ratio means that, at the end of the simulation (after 5000 training steps), each sensor with *evolution framework* has its communication activities reduced[35] by 60%, and it implies that sensor with the *evolution framework* is able to mitigate the energy spent on communication in a long-run.

6.3.3 Early Depletion

Figures including *fig-6.5*, *fig-6.6*, *fig-6.7* and *fig-6.8* have illustrated the statistic characters of distribution of energy consumption in sensor network. The higher the values of v_p and v_m are, the more difference exists in the energy consumption among different sensors, and this will lead to early depletion of network since a sensor with higher energy consumption will run out of battery energy earlier and make the network disconnected. *Fig-6.7(c)(d)*

[35] The implementation of *evolution framework* does require additional administrative data from neighbors. But as discussed, such additional effort has been exaggerated.

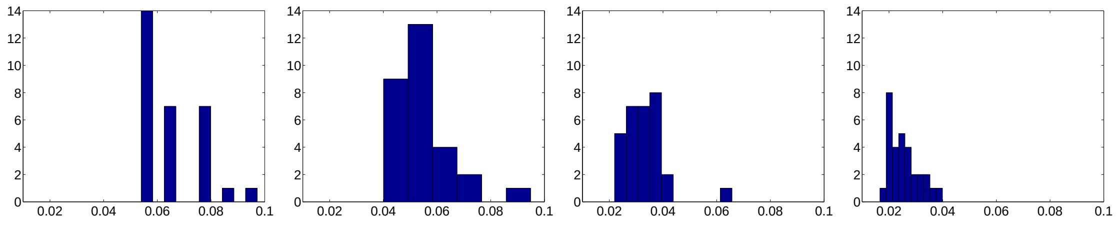


(a) "normal" (b) "EF+DC2" (c) "EF+DC4" (d) "EF+DC6"

Figure 6.12: *Histogram (over 30 simulations) of average amount of model inputs per sensor node. The X-axis presents the average amount, and the Y-axis indicates the corresponding occurrences. Simulations last* 5000 *training steps, and the maximum model complexity is set to* $\mathscr{C}_s = 100$.

and *fig-6.8(c)(d)* have shown the evidence of the advantage of "EF+DCi" schemes over "normal" scheme in terms of avoiding early depletion.

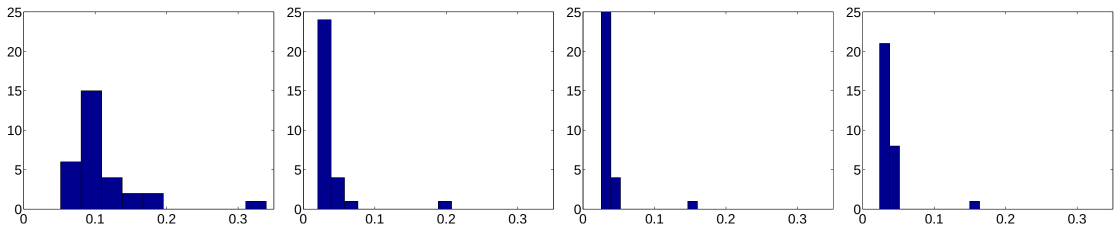
In another way, energy consumption during the training phase of one simulation is estimated by applying *eq-(6.14)* and *eq-(6.15)*. *Fig-6.13* depicts histograms (over 30 simulations) of the maximum communication-energy consumption in one sensor. *Fig-6.14* shows similar histograms about the energy consumption in computation. The X-axis, in"mAh" units, refers to the maximum amount $\mathscr{M}_{char}$ of electric charge consumed by all sensor nodes in the network; the Y-axis shows the respective occurrences. The smaller the value of $\mathscr{M}_{char}$ is, the less energy has been spent, and the longer life-time of sensor node will be.



(a) "normal" (b) "EF+DC2" (c) "EF+DC4" (d) "EF+DC6"

Figure 6.13: *Histogram (over 30 simulations) of the maximum value (over all sensors in the network) of communication-energy consumption. The X-axis, in "mAh" unit, refers to the maximum amount* $\mathscr{M}_{char}$ *of consumed electric charge in one simulation; the Y-axis shows the respective occurrences.*

It is manifested in *fig-6.13* and *fig-6.14* that the maximum energy consumption, in communication and computation respectively, spent in "EF+DCi" schemes is much less than the consumption in the "normal" scheme. In addition, as shown in *fig-6.13*, scheme "EF+DC6" delivers better and stable result in communication-energy consumption than the "EF+DC4" and "EF+DC2" schemes. This confirms the contribution of activity period $\mathscr{A}_p$ on communication-energy saving. In *fig-6.14*, since the *evolution framework* is the



(a) "normal" (b) "EF+DC2" (c) "EF+DC4" (d) "EF+DC6"

Figure 6.14: *Histogram (over 30 simulations) of the maximum value (over all sensors in the network) of computation-energy consumption. The X-axis, in "mAh" unit, refers to the maximum amount $\mathscr{M}_{char}$ of consumed electric charge in one simulation; the Y-axis shows the respective occurrences.*

primary factor reducing the computation effort, "EF+DCi" schemes have shown energy saving by factor of 40% comparing with the "normal" scheme.

Chapter 7

Summary and Conclusion

Distributed sensor networks built from “smart sensors” help to change the way human-being experience the physical world. “Smart sensors” have this name because of their abilities of data processing and wireless communication, even though these abilities are small compared to the average personal computer. Here more important, often these “smart sensors” have irreplaceable batteries, due to application requirements of autonomy and autarky as well as manufacture reasons.

Such “smart sensors” are often employed in diagnosis and control tasks, but in fulfilling such tasks they occasionally have to predict environment data and present the result in lieu of genuine information. Prediction is successful if the learning model, embedded in the sensor node, is stable and efficient enough to learn the dynamics on-line and from scratch, furthermore if the sensor nodes can communicate with each other, so that information required for building the learning model can be obtained also from neighbors. Realization of these assumptions is costly because (1) much computation is necessary for successfully training a learning model, and (2) frequent communication extends the sensor node’s duty cycle, thus reducing battery life. Sensor network application is therefore a compromise between prediction performance and energy consumption.

Contributions

Instead of improving communication protocol and network routing, the usual solution in such cases, this thesis proposes using an efficient embedded learning model and special operation schemes. These issues are addressed from two viewpoints, namely intra-node and inter-node solutions.

- The *intra-node* viewpoint focuses on *duty-cycle reduction*, replacing data transfer by data prediction with the aim of reducing costly communication. The implementation of the *duty-cycle reduction* can save energy by more than 40% when network’s connectivity is more than 5. This *duty-cycle reduction* is a success if an embedded model can be found which learns efficiently and stably from scratch, and which implements a precise prediction of the measurement data. A novel neural network structure, the spiral recurrent neural network (*SpiralRNN*), is such a model. *SpiralRNN* differs from conventional recurrent neural networks in the special structure of its hidden layer which possesses a trainable but structurally constrained recurrent layer, guaranteeing a bounded eigenvalue spectrum. Simulations of time series prediction have shown the efficiency and stability of the *SpiralRNN* model, which created the impulse for implementing the *duty-cycle reduction* scheme.

- The *inter-node* viewpoint focuses on the exchange of information among sensors in the neighborhood. The search for an embedded learning model had to be conducted in a systematic way, and for this we introduced the heuristic *evolution framework*, which is able to exchange information and modify the structure and parameters of the *SpiralRNN* model under the application constraints. The *evolution framework* is also able to dilute the communication pressure by changing the model's dependency on neighbor measurement, and omitting the communication effort for "unimportant" measurements. Depending on applications, this approach reduces the communication activity by 60% at long-run, and saves computation energy by more than 40%.

Achievements

We have implemented the *SpiralRNN* model in several simulations of time series prediction problems, comparing the *SpiralRNN* model with conventional neural network models such as a time delayed neural network, an echo state neural network, a simple recurrent net and a block-diagonal recurrent neural network. In all of those, the *SpiralRNN* model has been of excellent prediction performance. Equally, its stability has been demonstrated in simulations with benchmark time series including spike time series with period 21, Mackey-Glass time series and Lorenz time series. Particularly in the experiments with spike time series, the *SpiralRNN* model has shown its ability to store the temporal information with dependency on time longer than the size of its hidden layer. We have assessed the relationship between the short-term memory $\mathscr{M}_{st}$ and the number of hidden nodes $\mathbf{N}_h$ of a *SpiralRNN* model, which satisfy: $\mathscr{M}_{st} \simeq 16.8(\mathbf{N}_h - 2.0)^{0.42}$.

We have tested the *SpiralRNN* model in the physical world by implementing *Mouse-Tracking*, a toy software which analyzes the movements of a "mouse" device controlled by a human user. It has been shown that the *SpiralRNN* model was able to reproduce a figure-of-8 trajectory, which in general requires a learning model with substantial short-term memory.

The conditional prediction ability of the *SpiralRNN* model has been examined in a simulation of a warehouse logistics management scenario. It has been shown that the *SpiralRNN* model can simultaneously retain different dynamic models within the learning model and recall them when the sensor triggers unique information for various dynamics. Based on the simulations of conditional prediction, we also measured the associated memory $\mathscr{M}_a$ of *SpiralRNN*. The results have shown linearity with the number of hidden nodes $\mathbf{N}_h$ such that $\mathscr{M}_a \simeq 0.31\mathbf{N}_h$ for longer pattern ($l_p = 11$) and $\mathscr{M}_a \simeq 0.64\mathbf{N}_h$ for shorter pattern ($l_p = 5$), and both results are superior to that of the Hopfield network.

The prediction capability and special structure of the *SpiralRNN* model has encouraged the implementation of *duty-cycle reduction* and *evolution framework* in sensor network applications. We have simulated a heat diffusion scenario with sensor nodes spread out in a grid for predicting the temperature. *Duty-cycle reduction* and the *evolution framework* have been employed in a simulation where their impact on energy saving and on prediction performance was investigated. It was confirmed that *duty-cycle reduction* reduced the communication effort by replacing transferred data by predicted data from the sensor itself, and that the *evolution framework* scheme simplified the structure of the learning model by evolutionary operations, and with considerable confidence removed some "unimportant" input from the model, and hence saved energy on computation as well as on communication.

Conclusion

The novel *SpiralRNN* model has shown excellent prediction performance and stability. This formed the basis for the implementation of the *duty-cycle reduction* which curtails the communication effort, and the *evolution framework*, which mainly simplifies the computation. Both approaches aim to reduce the energy consumption of a sensor network at the expense of prediction performance, where the latter proved insignificant due to the effectiveness of the *SpiralRNN* model. The combination of *duty-cycle reduction* and the *evolution framework*, based on the learning model *SpiralRNN* and balancing performance against energy consumption, is therefore of great value in applications of sensor networks.

Appendix A

Eigenvalue Spectrum of *SpiralRNN*s

A.1 Preliminaries

Lemma 1 *Given value $n_{_{hnu}} \in \mathbb{N}^+$ and a nonsingular matrix $A \in \mathbb{R}_{n_{_{hnu}} \times n_{_{hnu}}}$ and matrix $D_A \in \mathbb{R}_{n_{_{hnu}} \times n_{_{hnu}}}$ as its canonical form, i.e. the diagonal matrix with eigenvalues $\{\lambda_1, \lambda_2, \cdots, \lambda_{n_{_{hnu}}}\}$ of A on the main diagonal, also given matrix $P_A = [\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_{n_{_{hnu}}}] \in \mathbb{R}_{n_{_{hnu}} \times n_{_{hnu}}}$ where columns of P_A are the unit eigenvectors of matrix A, then the eq-(A.1) holds. Matrix P_A is named the* modal matrix *of A, and matrix D_A is named the* spectral matrix *of A.*

$$A \;=\; P_A D_A P_A^{-1} \tag{A.1}$$

Lemma 2 *Given matrix $A, P_A, D_A \in \mathbb{R}_{n_{_{hnu}} \times n_{_{hnu}}}$ as defined in Lemma-1, the eq-(A.4) shows: (1) the* modal matrix *of exponential matrix A^k is identical with the modal matrix of matrix A; (2) the* spectral matrix *of exponential matrix A^k can be obtained by implementing an exponential function on the* spectral matrix *of matrix A.*

$$\begin{aligned} A^k &= \underbrace{P_A D_A P_A^{-1} \cdot P_A D_A P_A^{-1} \cdot P_A D_A P_A^{-1} \cdots P_A D_A P_A^{-1}}_{k} && \text{(A.2)}\\ &= P_A \underbrace{D_A \cdot D_A \cdot D_A \cdots D_A}_{k} P_A^{-1} && \text{(A.3)}\\ &= P_A D_A^k P_A^{-1} && \text{(A.4)} \end{aligned}$$

Lemma 3 *Given $\beta \in \mathbb{R}$, and matrix $A \in \mathbb{R}_{n_{_{hnu}} \times n_{_{hnu}}}$ with its* modal matrix *P_A and respective* spectral matrix *D_A, a matrix $B = \beta A$ will have the* modal matrix *P_B and the* spectral matrix *D_B, such that:*

$$\begin{aligned} P_B &= P_A \\ D_B &= \beta D_A \end{aligned}$$

Proof *According to Lemma-1, it holds* $A = P_A D_A P_A^{-1}$, *and*

$$\begin{aligned} B &= \beta A = \beta P_A D_A P_A^{-1} \\ &= P_A \beta D_A P_A^{-1} \\ \therefore BP_A &= P_A(\beta D_A) \end{aligned}$$

since matrix A's modal matrix P_A *is an orthogonal matrix and matrix* $D_B = \beta D_A$ *is a diagonal matrix like* D_A*, therefore matrix B's* modal matrix $P_B = P_A$ *and B's* spectral matrix $D_B = \beta D_A$. ▮

A.2 Eigenvalue Spectrum of Spiral Units

Definition For value $n_{_{hnu}} \in \mathbb{N}^+$, the *taxicab norm* of a vector $\vec{\beta} \in \mathbb{R}_{n_{_{hnu}} \times 1}$ is defined, using operator $||\cdot||_{taxi}$, as the sum of absolute values of all entries, *i.e.*:

$$||\vec{\beta}||_{taxi} = \sum_{i=1}^{n_{_{hnu}}} |\beta_i| = |\beta_1| + |\beta_2| + \ldots + |\beta_{n_{_{hnu}}}| \tag{A.5}$$

Theorem - 4 *Given value* $n_{_{hnu}} \in \mathbb{N}^+$*, and a matrix* $M \in \mathbb{R}_{n_{_{hnu}} \times n_{_{hnu}}}$ *which can be decomposed as*

$$M = \beta_1 \mathcal{P} + \beta_2 \mathcal{P}^2 + \ldots + \beta_{n_{_{hnu}}-1} \mathcal{P}^{n_{hnu}-1} = \sum_{i=1}^{n_{_{hnu}}-1} \beta_i \mathcal{P}^i$$

where vector $\vec{\beta} \in \mathbb{R}_{(n_{_{hnu}}-1)\times 1}$ *and matrix* $\mathcal{P} \in \mathbb{R}_{n_{_{hnu}} \times n_{_{hnu}}}$ *is the permutation matrix in eq-(3.4) which up-shifts the entries of multiplier for one position, the maximum absolute eigenvalue of matrix M is smaller than the* taxicab norm *of vector* $\vec{\beta}$*:*

$$\left|\lambda_{n_{_{hnu}}}\right| \leq \left|\left|\vec{\beta}\right|\right|_{taxi} = |\beta_1| + |\beta_2| + \ldots + |\beta_{n_{_{hnu}}-1}|$$

Proof Let matrices $P_\mathcal{P}$ and $D_\mathcal{P}$ respectively denote the modal matrix and the spectral matrix of matrix $\mathcal{P}$. Let matrices $P_{\mathcal{P}^i}$ and $D_{\mathcal{P}^i}$ respectively denote the modal matrix and the spectral matrix of matrix $\mathcal{P}^i$ with $i \in [1, n_{_{hnu}}-1]$, where matrix $\mathcal{P}^i$ is the exponentiation of $\mathcal{P}$ with the exponent i. According to *Lemma-1*, it holds:

$$\begin{aligned} \mathcal{P} &= (P_\mathcal{P})(D_\mathcal{P})(P_\mathcal{P})^{-1} \\ \mathcal{P}^i &= (P_{\mathcal{P}^i})(D_{\mathcal{P}^i})(P_{\mathcal{P}^i})^{-1}, \quad i \in [1, n_{_{hnu}}-1] \end{aligned}$$

According to *Lemma-2*, it satisfies:

$$\begin{aligned} P_{\mathcal{P}^i} &= P_\mathcal{P} \\ D_{\mathcal{P}^i} &= (D_\mathcal{P})^i \\ \text{with } i &\in [1, n_{_{hnu}}-1] \end{aligned}$$

Given matrix $M = \sum_{i=1}^{n_{hnu}-1} \beta_i \mathcal{P}^i$, it reads:

$$\begin{aligned} M &= \sum_{i=1}^{n_{hnu}-1} \beta_i (P_{\mathcal{P}^i})(D_{\mathcal{P}^i})(P_{\mathcal{P}^i})^{-1} \\ &= P_{\mathcal{P}} \left(\sum_{i=1}^{n_{hnu}-1} \beta_i (D_{\mathcal{P}^i}) \right) P_{\mathcal{P}}^{-1} \\ &= P_{\mathcal{P}} \left(\sum_{i=1}^{n_{hnu}-1} \beta_i \big(D_{\mathcal{P}}\big)^i \right) P_{\mathcal{P}}^{-1} \qquad (A.6) \\ with \qquad & \vec{\beta} \in \mathbb{R}_{n_{hnu} \times 1} \end{aligned}$$

Assuming the eigenvalues of matrix $\mathcal{P}$ is $\left\{ \hat{\lambda}_1, \cdots, \hat{\lambda}_{n_{hnu}} \right\}$, its spectral matrix $D_{\mathcal{P}}$ has $\left\{ \hat{\lambda}_1, \cdots, \hat{\lambda}_{n_{hnu}} \right\}$ in the diagonal. Therefore, with *eq-(A.6)*, matrix M's eigenvalue λ_k can be calculated as followed:

$$\lambda_k = \sum_{i=1}^{n_{hnu}-1} \beta_i (\hat{\lambda}_k)^i \quad \forall k \in [1, \cdots, n_{hnu}] \qquad (A.7)$$

As all eigenvalues of a permutation matrix lie in the unit cycle [81], it satisfies,

$$|\hat{\lambda}_k| = 1, \quad \forall k \in [1, \cdots, n_{hnu}] \qquad (A.8)$$

Thus, taking the absolute value over both side of *eq-(A.7)*, it holds $\forall k \in [1, \cdots, n_{hnu}]$, that:

$$\begin{aligned} |\lambda_k| &= \left| \sum_{i=1}^{n_{hnu}-1} \beta_i (\hat{\lambda}_k)^i \right| \leq \sum_{i=1}^{n_{hnu}-1} \left| \beta_i (\hat{\lambda}_k)^i \right| \\ &\leq \sum_{i=1}^{n_{hnu}-1} |\beta_i| \left| \hat{\lambda}_k^i \right| \\ &= \sum_{i=1}^{n_{hnu}-1} |\beta_i| \end{aligned}$$

Therefore, the maximum absolute eigenvalue of matrix M is bounded, and is limited by the *taxicab norm* of vector $\vec{\beta}$, *i.e.* $||\vec{\beta}||_{taxi}$.

A.3 Eigenvalue Spectrum of *SpiralRNN*s

Theorem - 5 *Let $N_{hn} \in \mathbb{N}$, $q \in [1, \cdots, N_{hn}]$, and a block-diagonal matrix $W \in \mathbb{R}_{N_{hn} \times N_{hn}}$ has $n_{units} \in \mathbb{N}$ numbers of sub-block matrices $M_k, k \in [1, \cdots, n_{units}]$, each of which can be* *decomposed as:*

$$M_k = \sum_{i=1}^{\{n_{hnu}\}_k - 1} \beta_i^{(k)} (\mathcal{P}_k)^i, \quad k \in [1, \cdots, n_{units}] \tag{A.9}$$

where $\{n_{hnu}\}_k$ *stands for the length of diagonal vector of sub-block matrix* M_k*,* $\beta_i^{(k)}$ *is the i-th entry of the vector* $\vec{\beta}^{(k)} \in \mathbb{R}_{(\{n_{hnu}\}_k - 1)\times 1}$*, matrix* $\mathcal{P}_k \in \mathbb{R}_{\{n_{hnu}\}_k \times \{n_{hnu}\}_k}$ *is the permutation matrix which up-shifts the entries of multiplier by one position,* $\mathcal{P}_k$ *has the same size as* M_k*. Note that* $\vec{\beta}^{(k)}$ *as a whole is a vector, and symbol* $^{(k)}$ *indicating the association with the k-th sub-block matrix is not an exponent.*

Then the maximum absolute eigenvalue $||\lambda||_\infty = \max\{|\lambda_q|\}$ *of matrix* W *is bounded by the maximum* taxicab norm *value in all* $\vec{\beta}^{(k)}$ *vectors, i.e.*

$$\begin{aligned} ||\lambda||_\infty &\leq \max_k \left\{ ||\vec{\beta}^{(k)}||_{taxi} \right\}, \quad k \in [1, \cdots, n_{units}] \\ \textit{where} \quad ||\vec{\beta}^{(k)}||_{taxi} &= \sum_{i}^{\{n_{hnu}\}_k - 1} |\beta_i^{(k)}| \end{aligned}$$

Proof *According to* Theorem-4*, the eigenvalue of the k-th sub-block matrix* M_k *of* W *is calculated as:*

$$|\lambda_j^{(k)}| \leq ||\vec{\beta}^{(k)}||_{taxi} = \sum_{i=1}^{\{n_{hnu}\}_k - 1} |\beta_i^{(k)}|, \quad \begin{array}{l} j \in [1, \cdots, \{n_{hnu}\}_k] \\ k \in [1, \cdots, n_{units}] \end{array} \tag{A.10}$$

Meanwhile, the determinant of block-diagonal matrix $\mathbf{det}(\cdot)$ *is equal to the product of determinants of all its sub-block matrices:*

$$\mathbf{det}(W) = \mathbf{det}(M_1) \times \mathbf{det}(M_2) \times \cdots \times \mathbf{det}(M_{n_{units}}) \tag{A.11}$$

Given $\lambda \in \mathbb{R}$ *and* $\mathbf{I}$ *an identity matrix with the same size as matrix* W*, the matrix* $(W - \lambda\mathbf{I})$ *is also a block-diagonal matrix. Let matrix* $\mathbf{I}_k$ *be an identity matrix with the same size of the k-th sub-block matrix* M_k *and* $\lambda_j^{(k)}$ *be the j-th eigenvalue of* M_k*,* $j \in [1, \cdots, \{n_{hnu}\}_k]$*, then the relations hold:*

$$\begin{aligned} \mathbf{det}(W - \lambda\mathbf{I}) &= \prod_{k=1}^{n_{units}} \mathbf{det}(M_k - \lambda\mathbf{I}_k) \\ &= \prod_{k=1}^{n_{units}} \Big((\lambda - \lambda_1^{(k)})(\lambda - \lambda_2^{(k)}) \cdots (\lambda - \lambda_{\{n_{hnu}\}_k}^{(k)}) \Big) \\ &= \prod_{k=1}^{n_{units}} \prod_{j=1}^{\{n_{hnu}\}_k} (\lambda - \lambda_j^{(k)}) \end{aligned}$$

Hence, the eigenvalue spectrum of a block-diagonal matrix aggregates the eigenvalue spectra of its sub-block matrices. Therefore, given any eigenvalue λ_j of block-diagonal matrix W, there exists one of the sub-block matrices which has the same eigenvalue, i.e. $\forall q \in [1, \cdots, N_{hn}]$, $\exists k \in [1, \cdots, n_{units}]$ and $\exists i \in [1, \cdots, \{n_{hnu}\}_k]$, such that:

$$|\lambda_q| = \left|\lambda_i^{(k)}\right| \leq \left|\left|\vec{\beta}^{(k)}\right|\right|_{taxi} = \sum_{j=1}^{\{n_{hnu}\}_k - 1} \left|\beta_j^{(k)}\right| \tag{A.12}$$

As a result, the eigenvalue spectrum of matrix W is limited by the maximum taxicab norm *value of the associated vectors.*

$$||\lambda||_\infty \leq \max_q \Big\{|\lambda_q|\Big\} \leq \max_k \Big\{||\vec{\beta}^{(k)}||_{taxi}\Big\}, \quad k \in [1, \cdots, n_{units}] \quad \blacksquare$$

Appendix B

MatLab Code for *SpiralRNN* Models

To implement *SpiralRNN* models in the MatLab environment, create MatLab scripts for all of the following functions and save them in a folder with the name "@spiralRNN".

- To start training a *SpiralRNN* model, go to the directory where folder "@spiralRNN" exists as a sub-folder.
- Construct the class "spiralRNN" by calling the constructor function "spiralRNN".

  $\boxed{\text{net = spiralRNN(dim,nuh);}}$
- In each time step with pair of network input and corresponding target: {dat,tar}, train the model with the command "trnng".

  $\boxed{\text{net = trnng(net,dat,tar);}}$
- Autonomous test can start, given the initial starting data "dat" and prediction step "fut", at any time with command "auttt".

  $\boxed{\text{output = auttt(net,dat,fut);}}$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net = spiralRNN(varargin)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%SPIRALRNN - Constructor of the MatLab class 'spiralRNN'
%  SPIRALRNN by itself only initializes the MatLab structure and
%  specifies the size of all its components and values of some
%  of them.
%  This function itself does not implement learning or testing, and such
%  operation will be conducted by other scripts in the same
%  class.
%
```

```
%  This script supports various ways of updating, which is controlled
%  by boolean parameters: 'trainDiff' and 'loopBack'. By default 'trainDiff'
%  is set to be 1 and loopBack = 0, note that 'loopBack' is active
%  only when trainDiff=1 .
%  - trainDiff=1, input of network remains unchanged whilst target
%    of output is set to the difference of data instead of
%    data itself, i.e. tar=x(t+1)-x(t) instead of tar=x(t+1).
%    Parameter 'loopBack' is active only when trainDiff=1
%    o loopBack=1, input will be set to as the sum of previous
%      input and previous output, i.e. x(t+1)=x(t)+y(t);
%    o loopBack=0, input will be set normal
%  - trainDiff=0, tar=x(t+1)
%
%  SYNTAX:
%
%    net = spiralRNN(dim, nhu, tdf, lpb) constructs the structure net
%    by specifying "dim" as the dimension of data fed into network, "nhu"
%    as number of hidden nodes in each hidden unit (by default, number
%    of hidden units will be equal to "dim"),
%    Parameters "tdf" and "lpb" are optional, their values will be assigned
%    to 'trainDiff' and 'loopBack' respectively, and their default values are
%    1 and 0.
%
%    It requires input:
%      dim - dimension of data;
%      nhu - number of neurons in each hidden units (number of hidden units
%            is equal to 'dim')
%      tdf - boolean argument, whether to train with difference of data
%      lpd - boolean argument, whether to loop back the output as input
%
%    and returns output:
%      net - object of class, containing all the information about
%            the neural network;
%

if or(nargin>4,nargin<2)
    error('Incorrect number of parameters')
else
    % dimension of data, which determinates the
    %number of input/output neurons in network
    dimension=varargin{1};

    % number of neurons in each spiral unit
    % (each spiral unit possesses same number of neurons)
    nhmini=varargin{2};
    if nargin>=3
        % to set the training target as the difference between data or not
        net.trainDiff=varargin{3};
    else
        net.trainDiff = 1;  % default value --- train the original data target
    end
    if nargin>=4
        net.loopBack = varargin{4};% to use the output value as in input or not
    else
        net.loopBack = 0;     % default value --- train without loopBack
    end
end

net.ninr = dimension; % dimension of input
net.nout = net.ninr; % dimension of output
net.nhmini = nhmini; % number of neurons in each spiral unit
net.nhidden = net.ninr*net.nhmini; % total number of neurons in hidden layer

outfn='linear'; hidfn='tanh';
net.outfn = outfn; % output activation function
net.hidfn = hidfn; % hidden activation function
net.MaxHidPropCoef = 1; % maximum absolute value of coefficient in W_hid

nl=0.01;
net.w1 = (2*rand(net.nhidden,net.ninr)-1)*nl;  % weights from input to hidden
net.b1 = (2*rand(net.nhidden,1)-1)*nl;         % bias of hidden neurons
net.w2 = (2*rand(net.nout,net.nhidden)-1)*nl;  % weights from hidden to output
net.b2 = (2*rand(net.nout,1)-1)*nl;            % bias of output neurons

% initial coef. in the hidden-weigth matrix
hidCoefIndx = [2:nhmini];
net.hidPropCoef = randn(length(hidCoefIndx),net.nout);
if length(hidCoefIndx)>1, deno=length(hidCoefIndx)-1;
else deno=1;
end
net.hidPropCoef = (2*rand(length(hidCoefIndx),net.nout)-1)/deno;

% construct the recurrent hidden weight matrix ---
% (permutation matrices in 'whidPack' will
% be used in construction of hidden matrix W_hid)
Mtem=diag(ones(1,nhmini-1),1)+diag(ones(1,1),-(nhmini-1));
for j=1:size(net.hidPropCoef,2)
    whidB = blkdiag(zeros((j-1)*net.nhmini),Mtem,zeros((net.nout-j)*net.nhmini));
    for i=1:size(net.hidPropCoef,1)
        packIdx=(j-1)*(nhmini-1)+i;
        net.whidPack{packIdx} = whidB^(hidCoefIndx(i)-1);
    end
end

net.input = zeros(net.ninr,1); % initial input
net.output = zeros(net.nout,1); % initial output
net.err = zeros(net.nout,1); % initial error
net.hidState = zeros(net.nhidden,1); % initial hidden state
net.preHidState = zeros(net.nhidden,1); % initial previous hidden state
net.preData = zeros(net.ninr,1); % initial previous data;

%  construct Vweight vector which assembles all the parameters to be trained.
%  This vector, instead of all other weight matrices (e.g. w1, w2, b1 etc.),
%  will be used in EKF; while those weight matrices will be used in
%  feed-forward step since it is more straight-forward
net.Vweight = [net.w1(:);net.b1(:);net.hidPropCoef(:);net.w2(:);net.b2(:)];

% total number of parameters/neural connection to be trained
net.noswei = length(net.Vweight);

% initial derivative
net.g_s_w = zeros(net.nhidden,net.noswei);%deri. of hidden state w.r.t. weights
net.g_o_w = zeros(net.nout,net.noswei); %deri. of network output w.r.t. weights

net.kalmanP = eye(net.noswei)*1; % initial P matrix for kalman filter
net.kalmanQ = eye(net.noswei)*1e-8; % initial Q matrix for kalman filter
net.kalmanR = eye(net.nout)*0.01; % initial R matrix for kalman filter

net.name = 'spiralRNN';
net = class(net,'spiralRNN');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [opt,hdv] = auttt(net,fin,nss)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%AUTTT - Implementation of autonomous testing in spiralRNN model.
%  AUTTT implements the autonomous test (i.e. output of previous step is
%  fed in as input "fin" for the current step, and so on) and returns the
%  result stream.
%
%  SYNTAX:
%
%    [opt,hdv] = auttt(net,fin,nss) returns the autonomous test results
%      for "nss" steps when net is triggered by the feed-in of "fin".
%      Returned results include the output "opt", hidden states "hdv"(for debug)
```

```
%
%    It requires input:
%      net - object of class, containing all the information about
%            the neural network;
%      fin - [dim,1] column vector which contains data to be fed in the
%            network as input
%      nss - total number of iterations in the autonomous prediction
%
%    and returns output:
%      opt - [dim,nss] matrix, contains sequence of output column vectors for
%            'nss' steps in autonomous prediction.
%      hdv - [nhn,nss] matrix, contains sequence of hidden-state vectors for
%            'nss' steps in autonomous prediction, where 'nhn' refers to
%            the number of hidden nodes in network
%

opt=zeros(length(fin),nss);
hdv=zeros(net.nhidden,nss);

for i=1:nss
    net=frwrd(net,fin);     % forward-step of network,
    if net.trainDiff == 1
        fin=net.output + fin;
    else
        fin=net.output;
    end
    opt(:,i)=fin;     % record output vectors from autonomous test
    hdv(:,i)=net.hidState;   % record hidden state vectors from autonomous test
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net = drvtv(net)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%DRVTV - Calculation of derivative
%  DRVTV calculates the derivative of output w.r.t. connection
%  weight. Error message has been stored in the component "err".
%
%  SYNTAX:
%
%    net = drvtv(net),
%
%    It requires input:
%      net - object of class, containing all the information about
%             the neural network;
%
%    and returns output:
%      net - object of class, the following components of net will be
%            directly modified by this script:
%        o g_s_w: [nhn,noswei] matrix refers to the derivative of hidden state
%                 w.r.t. all the weights, 'nhn' refers to number of hidden nodes
%                 in the network, 'noswei' is the total number of weights
%        o g_o_w: [dim,noswei] matrix refers to the derivative of output vector
%                 w.r.t. all the weights, 'dim' is dimension of data
%

% preparation of data
ninr=net.ninr;nh=net.nhidden;nout=net.nout;
maxCoef = net.MaxHidPropCoef;
tanhCoef = tanh(net.hidPropCoef); %connection weights
%  WHID = zeros(nh); % hidden-weight matrix
%  for i=1:length(tanhCoef(:)), WHID = net.whidPack{i}*tanhCoef(i) + WHID; end
WHID = fthdm(net);

% derivative of hidden FUNCTION
if or(strcmp(net.hidfn,'rectanh')==1,strcmp(net.hidfn,'tanh')==1)
    Vz2 = 1.0 - net.hidState.^2;
elseif strcmp(net.hidfn,'linear')==1
    Vz2 = ones(size(net.hidState));
else
    error('Unsupported hidden activiation function');
end
diagVz2 = diag(Vz2);

% GRADIENT CALCULATION
p_s_i = diagVz2*net.w1; % p_s_w: der. of current state w.r.t. network input
p_s_s = diagVz2*maxCoef*WHID; % der. of current state w.r.t. previous state
p_o_s = net.w2; % p_o_s: der. of output w.r.t. current state

% construct p_s_w: partial der. of current state w.r.t. weights
p_s_w = zeros(nh,net.noswei);lastidx=0;
p_s_w(:,lastidx+(1:nh*ninr)) = kron(net.input,eye(nh))'; % about net.w1
lastidx=lastidx + nh*ninr;
p_s_w(:,lastidx+(1:nh)) = eye(nh); % about net.b1
lastidx=lastidx + nh;
for i=1:length(tanhCoef(:)),
    p_s_w(:,lastidx+i)=maxCoef*(1-tanhCoef(i)^2)*net.whidPack{i}*net.preHidState;
end % about net.tanhCoef
lastidx=lastidx + length(tanhCoef(:));
```

```
p_s_w = diagVz2*p_s_w;

%  construct p_o_w: partial der. of output w.r.t. weights
%  only need to fill in those entries corresponding to the output
%  weight and the output bias
p_o_w = zeros(nout, net.noswei);
p_o_w(:,lastidx + (1:nout*nh)) = kron(net.hidState,eye(nout))'; % about net.w2
lastidx=lastidx + nout*nh;
p_o_w(:,lastidx + (1:nout)) = eye(nout); % about net.b2
lastidx=lastidx + nout;

% calculate gradient g_s_w: gradient of current state w.r.t. weights
net.g_s_w = p_s_w + p_s_s*net.g_s_w + p_s_i*net.g_o_w;

% calculate gradient g_o_w: gradient of output w.r.t. weights
net.g_o_w = p_o_w + p_o_s*net.g_s_w;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net = eKftn(net,drp)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%EKFTN - Implementation extended Kalman filter training algorithm
%  EKFTN - Implementation extended Kalman filter algorithm
%  on the training of network connection weights.
%
%  SYNTAX:
%
%    net = eKftn(net,drp) trains the values of connection weights
%      in net structure according to the derivative information in 'drp',
%      modification of value lies in the aggregated vector net.Vweight.
%      Matrix 'drp' is of size [dim,noswei]
%
%    It requires input:
%      net - object of class, containing all the information about
%            the neural network;
%
%    and returns output:
%      net - object of class, the following components will be directly
%            modified by this script:
%        o kalmanP: P matrix in extended Kalman filter
%        o kalmanR: R matrix in extended Kalman filter
%        o Vweight: vector which assembles all of the trainable weights
%
```

```
net.kalmanP=net.kalmanP+net.kalmanQ;

kal=net.kalmanP*transpose(drp)*inv((drp)*net.kalmanP*transpose(drp)+net.kalmanR);
net.kalmanP=net.kalmanP-kal*drp*net.kalmanP;% posterior covariance of weigths

net.Vweight=net.Vweight + kal*net.err;

% update of kalman R
maPeriodCoefR = 0.01; % coefficient for calculation of M.A. value of kalmanR
net.kalmanR = maPeriodCoefR*net.err*transpose(net.err) ...
              + (1-maPeriodCoefR)*net.kalmanR;

% Q is kept unchanged


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net = frwrd(net,fin)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%FRWRD - Implementation of forward step of 'spiralRNN' model.
%  FRWRD implements the iteration of spiralRNN model for one step
%  and return results. The network parameters are to be kept for
%  the sake of further iterations.
%
%  SYNTAX:
%
%    net = frwrd(net,fin), where fin is the fee-in vector,
%      returns the network with updated components
%
%    It requires input:
%      net - object of class, containing all the information about
%            the neural network;
%      fin - data vector which feeds in the network as input
%
%    and returns output:
%      net - object of class, the following components of net will be
%        directly modified by this script:
%        o input: input vector of network
%        o preHidState: hidden-state vector in the previous time step
%        o hidState: updated hidden-state vector in current time step
%        o output: output vector of network
%        o preData: given data of previous time step (only when net.loopBack=1)
%
```

```
% preparation
if net.loopBack==0,
    net.input = fin;
elseif net.loopBack==1
    net.input = net.output + net.preData;
    net.preData = fin;
end

% feed forward step
[output,HidState] = spiralFoward(net);

% restore data
net.preHidState = net.hidState;
net.hidState=HidState;
net.output=output;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [outState,hidState]=spiralFoward(net)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%SPIRALFORWARD - Implementation of feed forward step of spiralRNN.
%  SPIRALFORWARD implements the feed forward step of spiralRNN model
%  and returns the output vector "VoutStates" and hidden state vector
%  "VhidStates".

%  fetch the hidden-weight matrix
hdm = fthdm(net);

%  update the hidden state vector
hidState = net.hidState;
netin = net.w1*net.input  + net.b1;
netin = netin + hdm*hidState;
switch net.hidfn
    case 'tanh', hidState = tanh(netin);
    case 'linear', hidState = netin;
    otherwise, error('Unrecognize recfunc');
end

%  update of output vector
netin = net.w2*hidState + net.b2;
switch net.outfn
    case 'linear', outState= netin;
    case 'tanh', outState = tanh(netin);
    otherwise, error('Unrecognize outfunc');
```

```
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function hdm = fthdm(net)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%FTHDM - Obtain the hidden-weight matrix of 'spiralRNN' model
%  FTHDM can fetch the hidden-weight matrix of 'spiralRNN' model
%  directly. Creation of this script is mainly for reason of reviewing,
%  but it is also called by 'spiralFoward' which is a sub-routine
%  of script 'frwrd.m'.
%
%  SYNTAX
%
%    hdm = fthdm(net),
%
%    It requires input:
%      net - object of class, containing all the information about
%            the neural network;
%
%    and returns output:
%      hdm - hidden-weight matrix of 'spiralRNN'
%

%  recurrent coupling coefficients
tanhCoef = tanh(net.hidPropCoef);

%  initialization of matrix
hdm = zeros(net.nhidden);
for i=1:length(tanhCoef(:)),
    hdm = net.MaxHidPropCoef*net.whidPack{i}*tanhCoef(i) + hdm;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net = trnng(varargin)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%TRNNG - Implementation of training step in 'spiralRNN' model
%  TRNNG calculates the derivative of output w.r.t. to all
%  the synaptical weights in network and adjusts the values
%  of weights accordingly, finally returns back the trained
%  structure net.
%
```

```
%  SYNTAX:
%
%    net = trnng(net) trains the weights directly, assuming
%    the forward step has been already implemented and the error
%    message as well as hidden state vector etc are up-to-date
%    and available.
%
%    net = trnng(net,fin,tar) will conduct a forward step to
%    generate the output and calculate the error
%    at first and then implement the training and update of weights
%    of network.
%
%    It requires input:
%      net - object of class, containing all the information about
%            the neural network;
%      fin - data vector which feeds in the network as input;
%      tar - vector of target data;
%
%    and returns output:
%      net - object of class, the following components of net will be directly
%            modified by this script:
%        o err (only when number of input arguments is larger than 1)
%        o g_o_w (will be reset to zero only when loopBack=0)
%

%  fetch arguments
net = varargin{1};
if nargin==3,
    fin = varargin{2};
    tar = varargin{3};
    doForward = 1;
elseif nargin==1
else error('Number of input arguments is incorrect.')
end

% when required, implement feed-forward
if and(exist('doForward','var'),doForward==1),
    net = frwrd(net,fin);
    net.err = tar - net.output;
    if net.trainDiff == 1, net.err = net.err - fin; end
end

%  calculate the derivative of network output w.r.t. net connection weights
net = drvtv(net);
```

```
%  using KALMAN FILTER to adjust the values in 'Vweight'.
%  (In order to keep the interface of script 'eKftn.m' indentical
%  for all the recurrent neural network model, we do NOT simplify
%  the interface to net=eKftn(net). Anyway, it is trival)
net = eKftn(net,net.g_o_w);

% update network weights, (according to values of 'Vweight')
net = wgtup(net);

% reset the gradient information when necessary
if net.loopBack == 0
    % no gradient will be looped back when doing teacher forcing
    net.g_o_w = zeros(size(net.g_o_w));
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net = wgtup(net)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%WGTUP - Update of weights in all connection matrices.
%  WGTUP updates the value of all connection weights, given
%  the update component net.Vweight. Update order should be consistent
%  with the combination of the derivative information as in script
%  "drvtv.m"
%
%  SYNTAX:
%
%  net = wgtup(net),
%
%  It requires input:
%    net - object of class, containing all the information about
%          the neural network;
%
%  and returns output:
%    net - object of class, the following components of net will be directly
%          modified by this script:
%      o w1: weight matrix which connects input and hidden layer
%      o b1: bias vector of hidden nodes
%      o hidPropCoef: coefficient of connection among hidden nodes
%      o w2: weight matrix which connects hidden and output layer
%      o b2: bias vector of output nodes
%

Vweight=net.Vweight;
```

```
ninr=net.ninr;nh=net.nhidden;nout=net.nout;

% restore weight (input --> hidden)
lastidx=0;
net.w1(:) = Vweight(lastidx + (1:ninr*nh));
lastidx=lastidx + ninr*nh;

% restore bias of hidden neurons
net.b1(:) = Vweight(lastidx+(1:nh));
lastidx=lastidx+nh;

% restore hidden coefficient
net.hidPropCoef(:)=Vweight(lastidx+(1:prod(size(net.hidPropCoef))));
lastidx=lastidx+prod(size(net.hidPropCoef));

% restore weight (hidden --> output)
net.w2(:) = Vweight(lastidx+(1:nh*nout));
lastidx=lastidx+nh*nout;

% restore bias of output
net.b2(:) = Vweight(lastidx+(1:nout));
lastidx=lastidx+nout;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function res=get(net,ppn)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  GET - Fetch the value of component in given structuure. (for debug)
%  GET return the value res which belongs to the component with
%  the name ppn from structure net.
%
%  res=get(net,ppn)
%

res=eval(strcat('net.',ppn));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net=set(net,ppn,val)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%SET - Change of value of component in net (for debug)
%  SET enables outsider to change the configuration of structure
%  directly. This change of value will be permenent.
```

```
%
%  net=set(net,ppn,val) uses data val to reset the component
%  property, which has the name string as ppn, of structure net
%

eval(strcat('net.',ppn,'=val;'));
```

Appendix C

MatLab Code for *MouseTracking*

The *MouseTracking* is a toy software developed in the MatLab environment as the first application of the *SpiralRNN* model in real-world problems. The software measures the position of the mouse cursor on the computer monitor, preprocessing is applied to these raw measurement data such that those points dividing the trajectory path into even segments are treated as training data of the *SpiralRNN* model. These points separating stretches are called “boundary points”. Based on the training data, the *SpiralRNN* develops the embedded model to adapt to the dynamics of data so that it can make an autonomous prediction of the cursor trajectory, given its latest position.

The software supports keyboard commands with the keys “s”, “t”, “n” and “q”. Details of the keys can be found in the function description in the following. Note that one must keep the software interface on top (being activated or high-lighted) in order to make the software respond to keys being pressed.

To install, create MatLab scripts for all of the following functions and save them into the folder named “mousetracking” or other. In order to implement the *SpiralRNN* model, the folder “@spiralRNN” mentioned in *appendix-B* has to be placed as a subfolder in the aforementioned folder. To launch the software interface, go to the directory “mousetracking” in the MatLab environment and type the following command, where “nhdut” determines the size of the network model and “prdst” specifies the prediction length.

$\boxed{\text{net = mouseTrack(nhdut,prdst)}}$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function net = mouseTrack(nhdut,prdst)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  MOUSETRACK - implement MouseTracking which tracks the mouse cursor
%    MOUSETRACK - implement the MouseTracking, which constructs a SpiralRNN
%    model as the learning model, measures the pre-pocesses the data from
%    mouse position and uses these as training data. Training error
%    is drawn in the bottom subplot of interface, and upper subplot
%    depicts the measured position of cursor.
%    Autonomous test prediction can be lauched by human user at
%    any time, where a red curve will be drawn to show the prediction
%    trajectory of mouse cursor.
%
%    Keyborad command is supported, where "s" stands for starting the
%    training meanwhile user should move the mouse on certain imagine
%    trajectory, "t" stands for taking the autonomous test but it will
%    not stopped until the prediction steps has been reached,
%    "n" stands for start a new training and reinitial all the parameters,
%    "q" stands for quiting the software.
%    Software will stay in the pause mode after the autonomous test,
%    and wait for the next command either "s" or "t" or "n" or "q".
%
%    SYNTAX:
%      net = mouseTrack(nhdut,prdst)
%
%      It requires input:
%        nhdut - number of hidden nodes in each hidden unit of SpiralRNN
%        prdst - prediction step, number of iterations in autonomous test
%
%      and return output:
%        net - learning model of SpiralRNN
%

if (ischar(nhdut)),nhdut=str2num(nh_unit);end
if (ischar(prdst)),prdst=str2num(prdst);end

clear global CntKey
global CntKey

% initialization
dim=2;
net=spiralRNN(dim,nhdut); % construct the class
scrWidth=1152;scrHeight=864; % resolution of computer monitor
allowTest=0;
allowTrain=0;
iter=0;
trajectory=zeros(dim,5e3);
trajectErr=zeros(1,5e3);
lengthTraj=0;
lengthTraE=0;
%relaxing/pause time between each iteration, such that computer can relax
```

```
iterPause=0.01;

% preparation the interface
fig = gcf; clf; set(fig,'KeyPressFcn',@getCntKey);
figure(fig);subplot(2,1,1);axis([-1,1,-1,1]) ,title('Trajectory')

toContinue=1;
while toContinue
    iter=iter+1;

    % detect the key press
    allowTrain=0;
    allowTest=0;
    if CntKey=='s'
        allowTrain=1;
    elseif CntKey=='q'
        close all
        return;
    elseif CntKey=='t'
        allowTest=1;
    elseif CntKey=='n'
        clear class net
        iter=0;
        net=spiralRNN(dim,nhdut);
        fig = gcf; clf; set(fig,'KeyPressFcn',@getCntKey);
        lengthTraj=0;
        lengthTraE=0;

        CntKey='s';
        continue;
    else
    end

    if allowTrain==1

        % initialization
        if ~exist('utl','var'),
            utl = 0.02 ;
            dfd = utl;
        end
        if ~exist('knnpt','var')
            newdata=get(0,'PointerLocation');
            newdata(1)= (newdata(1)-scrWidth/2)/(scrWidth/2);
            newdata(2)= (newdata(2)-scrHeight/2)/(scrHeight/2);
            knnpt=transpose(newdata); % the lastest known point on the trajectory
            iter=iter-1;
            continue;
        end
        if ~exist('prevInP','var'),
            prevInP = knnpt; % the latest boundary point
        end

        trajectory(:,lengthTraj+1) = knnpt; % record the position of cursor
        lengthTraj=lengthTraj+1;

        % get data from mouse position
        fig=gcf;
        newdata=get(0,'PointerLocation');
        newdata(1)= (newdata(1)-scrWidth/2)/(scrWidth/2);
        newdata(2)= (newdata(2)-scrHeight/2)/(scrHeight/2);
        ltstm=transpose(newdata); % the latest measuremet of mouse cursor

        % interpolation
        if gtdst(knnpt,ltstm) >= dfd
            % when mouse cursor moves to a new position such that the distance
            % is far enough to form at least one new boundary point.

            % sgmpt - concatenation of boundary points
            % dfd - difficiency in length to new the next boundary point
            [sgmpt,dfd] = intpl(knnpt,ltstm,dfd,utl);
        else
            % when mouse cursor moves to a new position where the distance
            % is not far enough to form at least one new boundary point,
            % then update the value, skip the training and waiting
            % for next measurment.

            dfd = dfd - gtdst(knnpt,ltstm);
            iter=iter-1;
            knnpt = ltstm;
            continue;
        end

        % training
        nosIntPoint=size(sgmpt,2);
        for k=1:nosIntPoint
            % train the network
            net=trnng(net,prevInP,sgmpt(:,k));
            % record the training error
            iterErr(:,k) = get(net,'err');
            % assign current target as the input for next iteration
```

```
            prevInP=sgmpt(:,k);
        end

        % record the error and boundary points
        trajectErr((1:nosIntPoint)+lengthTraE)=mean(iterErr(:,1:nosIntPoint).^2);
        trajectory(:,(1:nosIntPoint)+lengthTraj)=sgmpt;
        lengthTraj = lengthTraj + nosIntPoint;
        lengthTraE = lengthTraE + nosIntPoint;

        % plot the trajectory and training error
        subplot(2,1,1),title('Trajectory')
        plot(trajectory(1,1:lengthTraj),trajectory(2,1:lengthTraj))
        drawnow
        if iter>=2
            subplot(2,1,2),
            plot(log(trajectErr(1:lengthTraE)/var(trajectErr(1:lengthTraE))));
            title('Logorithmic NMSE'),grid on, drawnow
        end

        % take the latest measurement point value in "knnpt"
        knnpt=ltstm;

        if size(trajectErr,2)>5e3,disp('TrnngIter>=5e3');return;end
        pause(iterPause); % slow down the simulation
    elseif allowTest==1
        % autonomous test
        testOutput=auttt(net,prevInP,prdst);

        % plot the result of autonomous test
        figure(fig)
        subplot(2,1,1),
        hold on,
        plot(testOutput(1,1),testOutput(2,1),'gp');
        for k=2:size(testOutput,2),
          plot(testOutput(1,1:k),testOutput(2,1:k),'r','linewidth',4); drawnow,
        end
        hold off
        title('Trajectory')

        % print out notice after plotting
        temStr = strcat(num2str(prdst),' steps');
        temStr = ['Red line indicates the autonomous prediction with ',temStr];
        disp(temStr);

        % release the indicator
        CntKey='';
        allowTest=0;
        iter=iter-1;
    else
        iter=iter-1;
    end

    pause(0.001) % relax the computer
end    % end of WHILE


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function dst = gtdst(pt1,pt2)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%GTDST - get the distance between two point (2-dimensional)
%  GTDST calculates the euclidean distance between two 2-dimensional
%  point, and return the value.
%
%  SYNTAX:
%    dst = gtdst(pt1,pt2);
%
%    It requires the input:
%      pt1 - position vector of point 1
%      pt2 - position vector of point 2
%
%    and returns output:
%      dst - euclidean distance between two given points
%

dst = sqrt(sum((pt1-pt2).^2));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [sgp,dfd] = intpl(psp,lms,dfd,utl)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  INTPL - interpolate the mouse trajectory
%    INTPL - divide the path from "psp" to "lms" into even segments
%    with each segement has length equals "utl", and concantenate
%    all the segement points (points dividing the segments)
%    and the defficiency of the last trial
%
%  SYNTAX:
%    [sgp,dfd] = intpl(psp,lms,dfd,utl);
```

```
%
%    It requires the input:
%      psp - position vector of the latest boundary point
%      lms - position vector of the latest measurement of mouse cursor
%      dfd - deficiency (of length) which was required
%            in the previous interpolation
%      utl - unit of length which is used to segment the path of cursor
%
%    and returns the output:
%      sgp - concantenation of the position of newly formed boundary points
%      dfd - deficiency (of length) which is required
%            in the current interpolation
%

% the distance from psp to lms
availableDist = gtdst(psp,lms);

if availableDist < dfd
    % when "availableDist" is not longer than "dfd"
    % to form a segmentation

    sgp = [];
    dfd = dfd - availableDist;  % update dfd
else
    % number of segments which can be made within the "availableDist"
    fac = ceil((availableDist-dfd)/utl);

    % construct the boundary point one by one, and store in "sgp"
    sgp = zeros(2,fac);
    sgp(:,1) = gnsgp(psp,lms,dfd);
    for i=2:fac
        sgp(:,i) = gnsgp(sgp(:,i-1),lms,utl);
    end

    % calculate the deficiency to form another boundary point
    dfd=utl - gtdst(sgp(:,fac),lms);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function nsp = gnsgp(psp,lms,rql)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  GNSGP - generate the boundary points
%    GNSGP generates one boundary point based on the position of previous
%    boundary point and the latest measurement point.
%
%  SYNTAX:
%    nsp = gnsgp(psp,lms,rql)
%
%  It requires the input:
%    psp - the previous boundary points
%    lms - position vector of the latest measurement of mouse cursor
%    rql - the required length in order to form a segment
%
%  and returns the output:
%    nsp - the position vector of the newly constructed boundary point
%

% distance
dPs = gtdst(psp,lms);

% calculate the position of boundary point
% such that distance from "psp" to "nsp"
% is equal to "rql"
nsp = zeros(2,1);
nsp(1) = rql/dPs*(lms(1)- psp(1))+psp(1);
nsp(2) = rql/dPs*(lms(2)-psp(2))+psp(2);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function getCntKey(src,evnt)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% get the key press
global CntKey
CntKey = evnt.Character;
```

List of Figures

List of Tables

Bibliography

[1] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, “Connecting the physical world with pervasive networks,” *IEEE Pervasive Computing*, vol. 1, pp. 59– 69, 2002.

[2] J. J. Garrahan, P. A. Russo, K. Kitami, and R. Kung, “Intelligent network overview,” *IEEE Communication Magazine*, vol. March, pp. 30–36, 1993.

[3] J. Stankovic, T. Abdelzaher, C. Lu, L. Sha, and J. Hou, “Real-time communication and coordination in embedded sensor networks,” *Proceedings of the IEEE*, vol. 91, pp. 1002–1022, 2003.

[4] C.-Y. Chong and S. Kumar, “Sensor networks: evolution, opportunities, and challenges,” *Proceedings of the IEEE*, vol. 91, pp. 1247–1256, 2003.

[5] A. Kun, W. Miller, and W. Lenharth, “Modular system architecture for electronic device integration in police cruisers,” *Intelligent Vehicle Symposium, 2002. IEEE*, vol. 1, pp. 109–114, 2002.

[6] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, “Wireless sensor networks for habitat monitoring,” *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA’02)*, 2002.

[7] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: A survey,” *Computer Networks*, vol. 38, pp. 393–422, 2002.

[8] G. J. Pottie and W. J. Kaiser, “Wireless integrated network sensors,” *Commun. ACM*, vol. 43, no. 5, pp. 51–58, 2000.

[9] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, and D. Estrin, “Building efficient wireless sensor networks with low-level naming,” in *SOSP ’01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, ISBN: 1-58113-389-8, (New York, NY, USA), pp. 146–159, ACM Press, 2001.

[10] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, “Next century challenges: Scalable coordination in sensor networks,” *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM ’99)*, pp. 263–270, 1999.

[11] A. Kansal, A. A. Somasundara, D. D. Jea, M. B. Srivastava, and D. Estrin, “Intelligent fluid infrastructure for embedded networks,” in *MobiSys ’04: Proceedings of the 2nd international*

conference on Mobile systems, applications, and services, (New York, NY, USA), pp. 111–124, ACM Press, 2004.

[12] E. C. Uberbacher and R. J. Mural, "Locating protein-coding regions in human DNA sequences by a multiple sensor-neural network approach," *Proceedings of the National Academy of Sciences, USA.*, vol. 88, pp. 11261–11265, 1991.

[13] E. Yoneki, "Evolution of ubiquitous computing with sensor networks in urban environments," in *Ubicomp - Workshop on Metapolis and Urban Life*, pp. 56–60, September 2005.

[14] B. Hong and V. K. Prasanna, "Constrained flow optimization with applications to data gathering in sensor networks," in *First International Workshop on Algorithmic Aspects of Wireless Sensor Networks*, 2004.

[15] A. Hac, *Wireless Sensor Network Designs.* John Wiley & Sons, 2003.

[16] R. V. Dyck and L. Miller, "Distributed sensor processing over an ad hoc wireless network: simulation framework and performance criteria," in *Military communications conference*, 2001.

[17] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 1567– 1576, 2002.

[18] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris, "Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks," *Wireless Networks*, vol. 8, pp. 481–494, 2004.

[19] V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves, "Energy-efficient, collision-free medium access control for wireless sensor networks," *Wireless Networks*, vol. 12, pp. 63–78, 2006.

[20] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, 1989.

[21] F. Scarselli, "Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results.," *Neural Networks*, vol. 11, pp. 15–37, 1998.

[22] A. Waibel, T. Hanazawa, G. Hinton, and K. Shikano, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37(3), pp. 328–339, 1989.

[23] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the theory of neural computation.* Addison Wesley, 1991.

[24] L. Li, "Approximation theory and recurrent networks," *International Joint Conference on Neural Network*, vol. 2, pp. 266–271, 1992.

[25] J. Choi, M. Bouchard, and T. Yeap, "Decision feedback recurrent neural equalization with fast convergence rate," *IEEE Transactions on Neural Networks*, vol. 16, pp. 699–708, 2005.

[26] A. C. Tsoi and A. Back, "Locally recurrent globally feedforward networks: a critical review of architectures," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 229–239, 1994.

[27] J. Perez-Ortiz, J. Calera-Rubio, and M. Forcada, "A comparison between recurrent neural architectures for real-time nonlinear prediction of speech signals," *Neural Networks for Signal Processing XI, 2001*, pp. 73–81, 2001.

[28] H. Zimmermann, R. Grothmann, A. Schaefer, and C. Tietz, *New Directions in Statistical Signal Processing: From Systems to Brain*, ch. Identification and Forecasting of Large Dynamical Systems by Dynamical Consistent Neural Networks. MIT Press, 2005.

[29] D. C. Psichogios and L. H. Ungar, "A hybrid neural network-first principles approach to process modeling," *AIChE Journal*, vol. 38, pp. 1499 – 1511, 2004.

[30] Y. Yao, G. Marcialis, M. Pontil, P. Frasconi, and F. Roli, "Combining flat and structured representations for fingerprint classification with recursive neural networks and support vector machines," *Pattern Recognition*, vol. 36, pp. 397–406, 2003.

[31] E. J. Hartman, J. D. Keeler, and J. Nowalski, "Layered neural networks with Gaussian hidden units as universal approximations," *Neural Computation*, vol. 2, pp. 210–215, 1990.

[32] K. Watanabe, J. Tang, M. Nakamura, S. Koga, and T. Fukuda, "Fuzzy-Gaussian neural network and its application to mobile robot control," *IEEE Transactions on Control Systems Technology*, vol. 4, pp. 193–199, 1996.

[33] A. Petrosian, D. Prokhorov, W. Lajara-Nanson, and R. Schiffer, "Recurrent neural network-based approach for early recognition of Alzheimer's disease in EEG," *Clinical Neurophysiology*, vol. 112, pp. 1378–1387, 2001.

[34] P. Angeline, G. Saunders, and J. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, pp. 54–65, 1994.

[35] J. Sum, C.-s. Leung, G. H. Young, and W.-k. Kan, "On the Kalman filtering method in neural-network training and pruning," *IEEE Transactions on Neural Networks*, vol. 10, pp. 161–166, 1999.

[36] J. Sum, L. wan Chan, C. sing Leung, and G. H. Young, "Extended Kalman filter-based pruning method for recurrent neural networks," *Neural Comput.*, vol. 10, no. 6, pp. 1481–1505, 1998.

[37] R. Setiono and H. Liu, "Neural-network feature selector," *IEEE Transactions on Neural Networks*, vol. 8, pp. 654–662, 1997.

[38] W. McCulloch and W. Pitts, *Bulletin of Mathematical Biophysics*, ch. A Logical Calculus of Ideas Immanent in Nervous Activity, pp. 115–133. Springer New York, 1988. Reprinted from theBulletin of Mathematical Biophysics, Vol. 5, pp. 115-133 (1943).

[39] R. P. Lippmann, "An introduction to computing with neural nets," *IEEE Acoustics, Speech and Signal Processing Magazine*, vol. 4, pp. 4–22, Apr. 1987.

[40] R. P. Lippmann, “Pattern classification using neural networks,” *IEEE Comm. Mag.*, vol. 27, pp. 47–69, 1989. November.

[41] J. Bioch, O. v. d. Meer, and R. Potharst, “Classification using Bayesian neural nets,” *IEEE International Conference on Neural Networks*, vol. 3, pp. 1488–1494, 1996.

[42] D. E. Rumelhart and e. a. J. L. McClelland, *Parallel distributed processing: explorations in the microstructure of cognition.* Cambridge, Mass.,MIT Press, 1986.

[43] T. Sejnowski and C. Rosenberg, “Parallel networks that learn to pronounce english text,” *Complex Systems*, vol. 1, pp. 145–168, 1987.

[44] J. Ma, “The capacity of time-delay recurrent neural network for storing spatio-temporal sequences,” *Neurocomputing*, vol. 62, pp. 19–37, 2004.

[45] A. D. Back and A. C. Tsoi, “FIR and IIR synapses, a new neural network architecture for time series modeling,” *Neural Computations*, vol. 3, pp. 375–385, 1991.

[46] J. L. Elman, “Finding structure in time,” *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[47] H. Zimmermann, R. Grothmann, A. Schaefer, and C. Tietz, “Dynamical consistent recurrent neural networks,” *Int. Joint Conference on Neural Networks (IJCNN)*, 2005.

[48] H. Jaeger, “The ”Echo State” approach to analysing and training recurrent neural networks,” Tech. Rep. GMD Report 148, German National Research Center for Information Technology, 2001.

[49] H. Jaeger, “Short term memory in echo state networks,” Tech. Rep. GMD Report 152, German National Research Center for Information Technology, 2002.

[50] H. Jaeger, “Adaptive nonlinear system identification with echo state networks,” *Advances in Neural Information Processing Systems*, vol. 15, pp. 593–600, 2003.

[51] H. Jaeger, “Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the ”echo state network” approach.,” Tech. Rep. GMD Report 159, German National Research Center for Information Technology, 2002.

[52] S. Sivakumar, W. Robertson, and W. Phillips, “Online stabilization of block-diagonal recurrent neural networks,” *IEEE Transactions on Neural Networks*, vol. 10, pp. 167 – 175, 1999.

[53] P. Mastorocostas and J. Theocharis, “On stable learning algorithm for block-diagonal recurrent neural networks, part 1: the RENNCOM algorithm,” *IEEE International Joint Conference on Neural Networks*, vol. 2, pp. 815– 820, 2004.

[54] L. Bottou, *On-line learning in neural networks*, ch. On-line learning and stochastic approximations, pp. 9–42. New York, NY, USA: Cambridge University Press, 1998.

[55] R. Kalman, “A new approach to linear filtering and prediction problems,” *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, pp. 35–45, 1960.

[56] F. Lewis, *Optimal Estimation: With an Introduction to Stochastic Control Theory.* A Wiley-Interscience Publication, 1986. ISBN: 0-471-83741-5.

[57] G. Welch and G. Bishop, "An introduction to the Kalman filter," Tech. Rep. Technical Report 95-041, University of North Carolina at Chapel Hill, Department of Computer Science, 2002.

[58] J. A. Perez-Ortiz, F. A. Gers, D. Eck, and J. Schmidhuber, "Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets," *Neural Network*, vol. 16, pp. 241–250, Mar 2003.

[59] P. Trebatickye, "Recurrent neural network training with the extended Kalman filter," *IIT.SRC 2005*, pp. 57–64, 2005.

[60] P. L. Bogler, "Tracking a maneuvering target using input estimation.," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-23, no. 3, pp. 298–310, 1987.

[61] S. Singhal and L. Wu, "Training multilayer perceptrons with the extended Kalman algorithm," *Advances in neural information processing systems 1*, pp. 133–140, 1989.

[62] D. E. Catlin, *Estimation, Control, and the Discrete Kalman Filter.* ISBN: 038796777X, Springer, 1988.

[63] Y.-R. Kim, S.-K. Sul, and M.-H. Park, "Speed sensorless vector control of an induction motor using an extended Kalman filter," *Industry Applications Society Annual Meeting, 1992., Conference Record of the 1992 IEEE*, vol. 1, pp. 594–599, 1992.

[64] R. G. Brown and P. Y. C. Hwang, *Introduction to Random Signals and Applied Kalman Filtering: with MATLAB excercises and solutions.* John Wiley & Sons,Inc., 3rd ed., 1997.

[65] L. Xie and Y. C. Soh, "Robust Kalman filtering for uncertain systems," *Syst. Control Lett.*, vol. 22, no. 2, pp. 123–129, 1994.

[66] R. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computtion*, vol. 1, pp. 270–280, 1989.

[67] A. W. Smith and D. Z. and, "Learning sequential structure with the real-time recurrent learning algorithm," *International Journal of Neural Systems*, vol. 1, pp. 125 – 131, 1989.

[68] T. Chow and Y. Fang, "A recurrent neural-network-based real-time learning controlstrategy applying to nonlinear systems with unknown dynamics," *IEEE Transactions on Industrial Electronics*, vol. 45, pp. 151–161, 1998.

[69] C. Fi-John, C. Li-Chiu, and H. Hau-Lung, "Real-time recurrent learning neural network for stream-flow forecasting," *Hydrological processes*, vol. 16, pp. 2577–2588, 2002.

[70] S. Haykin, ed., *Nueral Networks - a Comprehensive Foundation.* Prentice Hall, 1999.

[71] N. B. Toomarian and J. Barhen, "Learning a trajectory using adjoint functions and teacher forcing," *Neural Networks*, vol. 5, pp. 473 – 484, 1992.

[72] N. Toomarian and J. Barhen, "Fast temporal neural learning using teacher forcing," *International Joint Conference on Neural Network*, vol. 1, pp. 817–822, 1991.

[73] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra-low power wireless research," in *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors*, pp. 364–369, 2005.

[74] T. Bäck, F. Hoffmeister, and H.-P. Schwefel, "A survey of evolution strategies," *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 2–9, 1991.

[75] H.-G. Beyer, "Evolution strategies." http://www.scholarpedia.org/, September 2007.

[76] R. K. Belew, J. McInerney, and N. N. Schraudolph, *Artificial Life II*, ch. Evolving Networks: Using the Genetic Algorithm with Connectionist Learning, pp. 511–547. Addison-Wesley, 1992.

[77] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," *Evolutionary Computation*, vol. 4, pp. 2588 – 2595, 2003.

[78] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, pp. 99–127, 2002.

[79] K. O. Stanley, *Efficient Evolution of Neural Networks through Complexification.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2004.

[80] H. Gao, R. Sollacher, and H.-P. Kriegel, "Spiral recurrent neural network for online learning," in *15th European Symposium On Artificial Neural Networks Advances in Computational Intelligence and Learning*, (Bruges (Belgium)), April 2007.

[81] K. Wieand, "Eigenvalue distributions of random permutation matrices," *The Annals of Probability*, vol. 28, no. 4, pp. 1563–1587, 2000.

[82] D. Geller, I. Kra, S. Popescu, and S. Simanca, "On circulant matrices." Preprint.

[83] R. M. Gray, *Toeplitz and Circulant Matrices: A Review.* Now Publishers, Norwell, Massachusetts, 2006.

[84] B. Schuermann, J. Hollatz, and U. Ramacher, "Models of brain function and artificial neuronal nets," *Models of Brain Function and Artificial Neuronal Nets*, pp. 167–185, 1990.

[85] L. Glass and M. C. Mackey, *From Clocks to Chaos, The Rhythms of Life.* Princeton University Press, 1988.

[86] H. Gao and R. Sollacher, "Condictional prediction of time series using spiral recurrent neural network," in *European Symposium on Artificial Neural Networks Advances in Computational Intelligence and Learning*, 2008.

[87] R. Sollacher and H. Gao, "Efficient online learning with spiral recurrent neural networks," in *to appear in: International Joint Conference on Neural Networks*, 2008.

[88] R. J. McEliece, E. C. Posner, E. R. Rodemich, and S. S. Venkatesh, “The capacity of the hopfield associative memory,” *IEEE Trans. Inf. Theor.*, vol. 33, no. 4, pp. 461–482, 1987.

[89] R. Wattenhofer, L. Li, P. Bahl, and Y.-M. Wang, “Distributed topology control for power efficient operation in multihop wireless ad hoc networks,” in *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, 2001.

[90] W. Krause, R. Sollacher, and M. Greiner, *Lecture Notes in Computer Science*, ch. Self-* Topology Control in Wireless Multihop Ad Hoc Communication Networks, pp. 49–62. Springer Berlin / Heidelberg, 2005.

[91] L. Kleinrock and J. Silverster, “Optimum transmission radii for packet radio networks or why six is a magic number,” in *National Telecommunications Conference*, 1978.