Master's Thesis

Information Filtering System with Reliability-ranked Agents

1410208 Tran Xuan Hoang

Supervisor : Professor Satoshi Tojo
Main Examiner : Professor Satoshi Tojo
Examiners : Associate Professor Minh Le Nguyen
Associate Professor Kiyoaki Shirai

School of Information Science
Japan Advanced Institute of Science and Technology

August 2016

Information Filtering System with Reliability-ranked Agents

Tran Xuan Hoang (1410208)
School of Information Science
Japan Advanced Institute of Science and Technology

June 29, 2016

Keywords: information filtering, artificial neural networks, probabilistic reasoning, multi-agent system.

Today, information overload has become a serious problem with the explosive growth of resources available on the Internet. Web users are commonly overwhelmed by huge amount of data and are faced with the challenge of finding the most relevant and reliable information in a timely manner. It is critical to use intelligent software systems to assist users in finding the right information from an abundance of Web pages. Furthermore, if researchers can quickly find out, from an article database, appropriate research papers and/or academic journals which they believe that they should take time to read just by using intelligent software, their time for searching will be saved.

In order to solve the problem, this research proposes a model of multiple agents in which they gather data, exchange it together, and be ranked by the process of analyzing reliability using probabilistic reasoning. Belief and reliability among agents will be changed during the time they interact with each other. The process of ranking and classifying each agent is able to be trusted or not, therefore, are meaningful since they help us find out which one we should believe. From that result, important decisions can be made.

Basically, the significance of this research can be divided into two main points. Firstly, this research provides us a method of classifying agents as reliable or untrustworthy. This classification method is a good technique for solving a wide range of problems. Secondly, a concrete implementation of an information filtering system will demonstrate the advantages of flexible application of probabilistic reasoning in analyzing the reliability of agents as well as the application of neural networks in extracting useful information. Using the system to download and filter webpages and/or articles, all what users see are articles that really interest him/her. This research will also have a variety of applications for solving other problems such as predicting game strategies, finding malfunction parts, analyzing detective stories, etc.

This research aims to obtain two main goals. Firstly, the research aims to find a method of analyzing reliability of intelligent agents for ranking them when they exchange information together. In a simple view point, agents are intelligent autonomous computer programs (software). Autonomous software can be independent of human's control and has the ability to make its own decisions while intelligent software includes some qualities that the human mind has, such as the ability to understand language, to recognize picture, to solve problem and to learn. The second goal of this research is to construct an information filtering system that contains multiple intelligent agents, ranks them bases on their reliability and filters out useful data (find out reliable and trustworthy health information) gathered and exchanged by them using a combination of statistical probability and neural networks.

Acknowledgements

I would like to acknowledge and thank Professor S. Tojo, Doctor K. Sano and the members of Tojo-laboratory in the School of Information Science, JAIST that assisted me with this research. Their continued support, excitement and willingness to provide feedback made the completion of this research an enjoyable experience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Information Filtering System

Information filtering system appeared as the information or data available on the Internet became explosive and the web users' demand on filtering out unnecessary information while keeping relevant information became huge enough that data scientists started to conduct the very beginning studies on this topic. Information filtering usually sticks to user profiles - that is users give the filtering system profiles of descriptions showing their interests, tastes or preferences on information they expect and the system bases on those descriptions will gather and select and present to them only information that is seen as matching and relevant.

1.1.2 Previous Work on Filtering

The following table briefly summarizes some of typical information filtering systems and frameworks.

Table 1.1: Typical information filtering systems and frameworks. In the **Author** column, only one author name is listed for each filter.

Year	Filter	Motivation	Author
1970s	SMART	Vector Space Model	Salton
1977	FRUMP	Natural Language	DeJong
1985	RUBRIC	Probabilistic Logic	Cooper
1991	FERRET	Genetic Algorithms	Mauldin
1995	SIFT	Vector Space Model	Yan
1996	InRoute	Bayesian Network	Callan
1997	*framework*	Fuzzy Model	Miyamoto
1998 - 2005	LSI	Vector Space Model	Deerwester
2008	*framework*	Vector Space Model	Zimmer
2016	*framework*	Bayesian Linear	Bangrui

The above of course does not list all types of information filtering systems and frameworks that have been studied by scientist community, but in a certain aspect we see the lack of neural networks in the field of information filtering. In fact, neural networks appear in some work relating to information retrieval that is very close to information filtering.

However, applying neural networks to the field is still an open direction. In this research, we put one footstep in finding how useful it would be if we apply neural networks in the information filtering field. In addition, not only neural networks but the paradigm of ranking agents will also be investigated through this research.

1.2 Goals

This research aims to obtain two main goals

Goal 1: construct single intelligent agents that are capable of scoring articles' interestingness. Detailed design of a backpropagation neural network and a self-organizing map neural network for calculate the interestingness must be clearly provided.

Goal 2: define a mechanism of judging reliability of agents when they connect, exchange, cooperate and compete in a group. This mechanism must not depend on how agents act, or in other words it should not be affected by changes to obtain *Goal 1*.

1.3 Thesis Structure

The thesis is divided into five chapters: *Introduction*, *Background*, *Proposed System*, *Experiments* and *Conclusions and Future Directions*.

Chapter 2 introduces and clarifies the very basic concepts that will be used extensively in the subsequent chapters, and surveys some of the most popular filtering methods.

Chapter 3 describes the general model of the information filtering system as well as gives a concrete example about the model. Two types of neural network, the backpropagation neural network (BPNN) and the self-organizing map (SOM), are explained in detail, namely their architecture, mathematical foundation, algorithm. These two neural networks play an important role in the software application developed.

Chapter 4 is about conducting experiments and main finding. The process of generating training datasets will first be explained, then the comparison with previous method is shown.

Chapter 5 summarizes main contributions of this research and mentions some future directions. *Appendix A* describing the training datasets and *Appendix B* giving a short introduction to the software application developed are located at the end of the thesis.

Chapter 2

Background

In this chapter, basic concepts that are used throughout this thesis are explained. We do not deepen our understanding on these concepts but clarify them briefly so that reading the next chapters of the thesis will be easier. A brief summary of popular methods of information filtering will be explained at the end of this chapter.

2.1 Information Filtering and Information Retrieval

Information filtering and information retrieval are terminologies usually applied in connection with text information. Although both tasks have some common goals of obtaining the right information, they are however different in some aspects that are summarized as the following table

Table 2.1: Information filtering versus information retrieval.

	Information Filtering	**Information Retrieval**
Purpose	Filter out irrelevant data, recommend users only information that is relevant to them	Selecting relevant information from database(s) for users' queries
Based on	Descriptions of individual or group preferences, often called *profiles*	Sentence(s) describing in the form of questions, often called *queries*

In short, information filtering is about processing a stream of coming information to match users' set of likes, tastes and preferences while information retrieval is about answering immediate queries from a library of available information. In this thesis we are dealing with information filtering problem.

2.2 Intelligent Agents and Multi-agent Systems

In this thesis, we use the term *intelligent agents* to refer to intelligent computer programs that are able to filter information. The computer programs can run independently or cooperate and compete in a group. If one of these computer programs runs independently, it is called a single agent. In the case a group of computer programs connect in a network, exchange data together with the common goal of presenting relevant information to users, they form a *multi-agent system*. Cooperating helps the agent system enhances the diversity of information sources gathered, while competing actively promotes the agents' accuracy and reliability.

2.3 Supervised Learning, Unsupervised Learning and Artificial Neural Networks

In *supervised learning*, the agent learns from a set of given training datasets

$$\mathcal{T} = \{(e, e_{\text{label}}) | e \in \text{ a set of sampling examples}\}$$

so that when it receives a new example e^{new} that is not specified label yet, it will be able to find an example $e^0 \in \mathcal{T}$ that is the most similar with e^{new} and labels the new example as e^0_{label}.

In contrast to supervised learning, in *unsupervised learning* the agent is provided with an unlabeled set of sampling examples

$$\mathcal{U} = \{e\},$$

and tries to detect potentially useful clusters of these sampling examples. When the agent receives a new example e^{new}, it will use the trained knowledge to cluster the new example into one of groups of sampling examples to which e^{new} is mostly similar.

Artificial neural networks have been applied in a variety of science areas ranging from image or speech recognition to gene prediction and cancer classification. They are inspired by research on neural cells' sensory processing in human brains. An artificial neural network normally simulates the network of human brain neurons in a computer and is trained by algorithms that imitate the sensory processing so that it can *learn* to solve many types of problems [14]. Backpropagation and self-organizing map are the two network types that will be investigated in this thesis. The former as illustrated on the left hand side in fig. 2.1 is trained by a supervised learning algorithm in which all the weights are initialized to small random values. For each sampling example e, the network returns an output e_{real}. The squared difference between e_{real} and e_{label} is then computed. The sum of all these numbers over all sampling examples is called the *total error*. The smaller the total error is, the more accurate the network is. Therefore, this total error is propagated backward through the network to adjust the weights so that it will gradually become smaller. The self-organizing map as depicted on the right hand side is trained by an

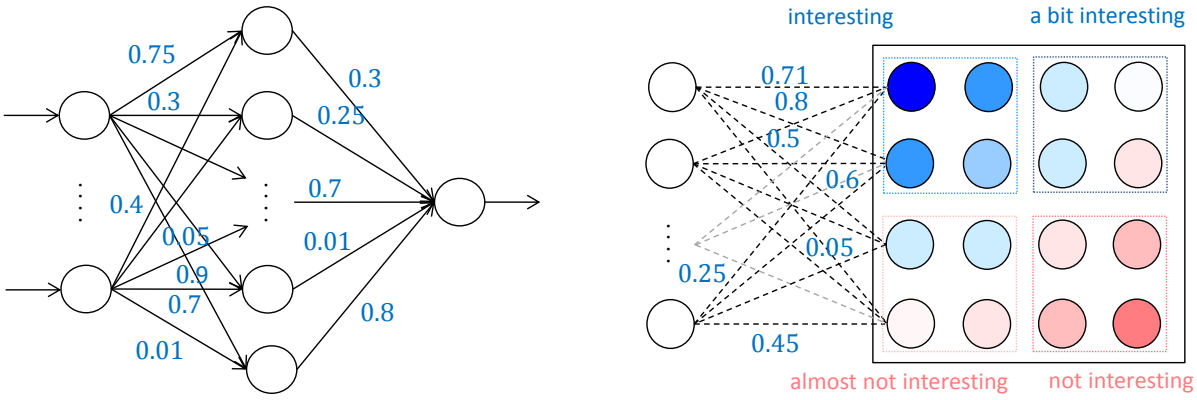


Figure 2.1: Backpropagation neural network and self-organizing map. Circles are neurons. Arrow and dashed lines are links between neurons, numeric values are weights associated with links.

unsupervised learning algorithm in which weights are adjusted so that similar sampling examples will be mapped onto nearby neurons that form a sub-map.

2.4 Information Filtering in a Nutshell

In this section, we survey some popular methods which has been widely studied and proved as the most successful filtering paradigms.

2.4.1 Vector Space Model

Salton in [25] proposed the *vector space model* (SVM) that was originally applied in information retrieval. The SVM after that was applied in information filtering task and have become the most popular method in both textual information retrieval and information filtering. In the SVM, each document $\mathbf{d}_i$ (may be seen as a piece of information) and the query $\mathbf{q}$ for searching the most relevant (may be considered as useful or interesting) document are represented in the form of vectors

$$\mathbf{d}_i = [w_{i1}, w_{i2}, \ldots, w_{iN}]$$
$$\mathbf{q} = [w_{q1}, w_{q2}, \ldots, w_{qM}]$$

where w_{in} is the weight of term (keyword or single word) n^{th} in document $\mathbf{d}_i$ and w_{qm} is the weight of term (keyword or single word) m^{th} in the query $\mathbf{q}$. The idea is that $\mathbf{q}$ is now considered as a very short document and the two vectors are normalized to have the same size so that the set of documents $\mathbf{d}_i$ and the query $\mathbf{q}$ can be evaluated in the same multiple-dimensional space. The *cosine similarity* between the query vector and a document vector become the measure of score of the document for that query

$$score(\mathbf{d}_i, \mathbf{q}) = cos(\mathbf{d}_i, \mathbf{q}) = \frac{\mathbf{d}_i \cdot \mathbf{q}}{\|\mathbf{d}_i\|\|\mathbf{q}\|}$$

where the nominator is the dot product of two vectors, and $\|.\|$ is the vector length operator. The higher the cosine value is, the more similar the document and the query are. Therefore, in order to filter out a set of highly relevant documents, we compute the cosine similarities between the query vector and each document vector in the collection, sorting the resulting scores and selecting the top-score documents.

The SVM is effective if the length of the documents and the document collection's size are small because with a large document, the dot production the the document vector and the query vector may require ten of thousands of arithmetic operations - that makes the SVM become not reasonable for the filtering task. To overcome this problem, Deerwester et al. in [6] proposed a method called *latent semantic indexing* in which both the document and query vectors are casted into lower-rank (smaller size) vectors enabling us to compute the similarity score with reasonable cost but still ensure accuracy.

2.4.2 Bayesian Network

The basic idea of using *Bayesian network* in filtering information is that the nodes of the network may represent topics and beliefs suggests what information is placed at a certain node. Degree of beliefs combining with the inference rule of the network provides a systematic mechanism of selecting information related to a particular topic. Callan is one of pioneers on this direction with the work of high speed document filtering by using inference networks [5]. He constructed the InRoute document filtering system that computes the degree of belief $\text{bel}_{term}(t)$ on how much each term t in a document contributes into the relevant between the document and the user's keyword profile using the *term*

frequency-inverse document frequency tf-idf defined as follows

$$\begin{aligned} \text{bel}_{term}(t) &= 0.4 + 0.6 \cdot ntf \cdot idf \\ ntf &= 0.4 + 0.6 \cdot \frac{tf}{tf + 0.5 + 1.5 \cdot \frac{dl}{avgdl}} \\ idf &= \frac{log\left(\frac{C + 0.5}{df}\right)}{log(C + 10)} \end{aligned}$$

where

tf	is the frequency of term t in the document,
dl	is the document legnth,
$avgdl$	is the average document length in the document set,
C	is the number of documents in the document set, and
df	is the number of documents in which term t appears.

2.4.3 Fuzzy Model

The idea of using fuzzy model in filtering information was proposed by Miyamoto as he extended it from his original idea of using fuzzy set theory to design a common framework for information retrieval [19, 20]. The information filtering systems in practice usually have to consider more than one criterion when gather and filter information. In such case, one mathematical model is not sufficient to deal with the accuracy of evaluating these criteria. Since fuzzy logic and fuzzy set theory provide us a general method to tackle this accuracy keeping problem, it is natural to apply them into the filtering task. Representing text in the form of fuzzy linguistic variables is also a flexible way to compute its similarity or relevant with respect to user's profile.

Chapter 3

Proposed System

In this chapter, we start with the main design for the framework of filtering information using neural networks and probabilistic reasoning.

3.1 General Model of the System

The information filtering system proposed in this thesis contains a set of agents cooperating in a network as in fig. 3.1. Recall that, in this thesis *information* is understood as *research articles*, and our agent-based system performs the task of filtering information means it will collect and present only the articles that the user may find highly interesting or useful.

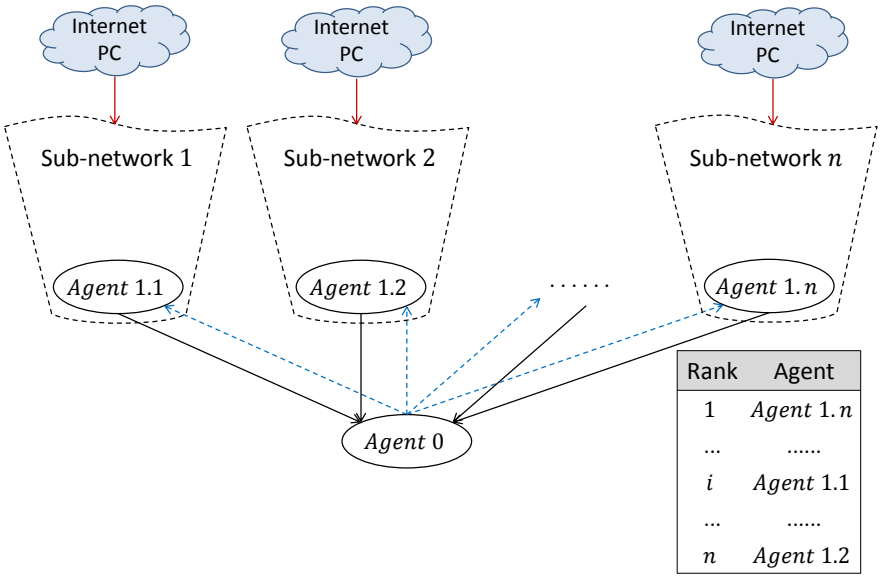


Figure 3.1: Proposed model of the information filtering system. Connecting in a tree-structured network, lower level agents send (represented as blue dashed arrows) their list of keywords that specify what kind of articles they want to search to higher level agents. Leaf-node agents gather (represented as dark red arrows) articles from the Internet or personal computers (PCs), the other non-leaf-node agents receive (represented as black dotted arrows) articles from their higher level agents that have been receiving keywords directly from them. Each lower level agent, after receiving articles from higher level agents using its own knowledge, will score those articles again to retrieve only the ones that have high interestingness and usefulness, and rank the sending agents in reliability of providing high score articles (i.e. *Agent* 0 may rank the higher level agents and creates a ranking table as above).

In order to suggest the user which articles are interesting and useful, we proposed the model of agent system (fig. 3.1 page 7) which conducts the following steps:

1. *Agent* 0 first sends (picturized as blue dashed arrows) a list of keywords given by user to all agents (*Agent* 1.1, *Agent* 1.2, ..., *Agent* 1.n) at the higher nodes specifying what research article topics she wants to find.

2. After receiving the list of keywords from *Agent* 0, *Agent* 1.1 in Sub-network 1 may add extra keywords based on its knowledge, then sends the extended list of keywords to all higher level agents that connect directly to it. If there is no other agent connecting to *Agent* 1.1, she becomes a leaf-node agent (agent who does not send keywords to any other agents). *Agent* 1.2, ..., *Agent* 1.n perform the same task as *Agent* 1.1 in Sub-network 2, ..., Sub-network n, respectively.

3. The receive-add-and-send-keyword process in step 2 propagates upward until it reaches agents at the leaf nodes of the tree-structured network.

4. Each leaf-node agent gathers (illustrated as dark red arrows) articles from the Internet or from the personal computer that she runs on, then uses its own knowledge to select only articles that are in its interests and could be interesting and useful with respect to the lower level agents as well.

5. Each non-leaf-node agent receives (illustrated as black dotted arrows) articles from all higher level agents that received keywords directly from it, then scores again those articles based on its knowledge and selects only the highly-scored ones to send to the agent in the lower level.

6. The receive-score-select-and-send-article process in step 5 propagates downward until it reaches *Agent* 0 at the root node. *Agent* 0 finally selects a set of articles that were considered as highly interesting at the higher level agents and at *Agent* 0 itself to recommend the user that those articles are worth reading.

7. All agents in the network will continue to exchange keywords and articles as in steps 2 and 5 asynchronously - that is, for example, not all agents in the same level receive articles or keywords from other agents at the same time, the numbers of times two higher level agents send articles to the same lower level agent are not necessary to be equal. While the articles are being exchanged between agents, each agent in the lower level ranks the agents in the higher level that directly send article to it and creates a table representing reliability degrees of these higher level agents based on their numbers of sent articles that are highly scored as interesting by the lower level agent. For example, *Agent* 0 after checking how many articles sent from *Agent* 1.1, *Agent* 1.2, ..., *Agent* 1.n are evaluated as interesting will create a reliability updating table similar to the one in the bottom right corner of fig. 3.1. The reliability updating tables will be used as a basic of selecting articles from and providing feedback for the higher level agents.

Each agent in the network may operate entirely automatically or semi-automatically in case she is controlled and utilized by a user. Because users may have different interests in reading research articles, each agent in the network is designed to has one neural network for capturing its user's preferences. Our network of agents becomes a system retrieving articles that are in the common interest of a majority of agents. If all agents in the network are possessed, used and updated by users, a network of exchanging and filtering research papers for researcher will be formed.

3.2 An Example of Agent Network

In this section, a concrete example will be introduced with the purpose of making the general model introduced in the previous section easier to understand. Imagine six researchers major in six research areas as in fig. 3.2:

- Logic
- Public announcement logic
- Dynamic epistemic logic
- Dynamic logic
- Machine learning
- Information filtering system with reliability-ranked agents

that share some common knowledge but are perspectively different. Each researcher trains one agent so that it can recognize the researcher's interests in his/her expertise area. Researchers will exchange research articles together by means of these agents as support tool of scoring and filtering gathered articles.

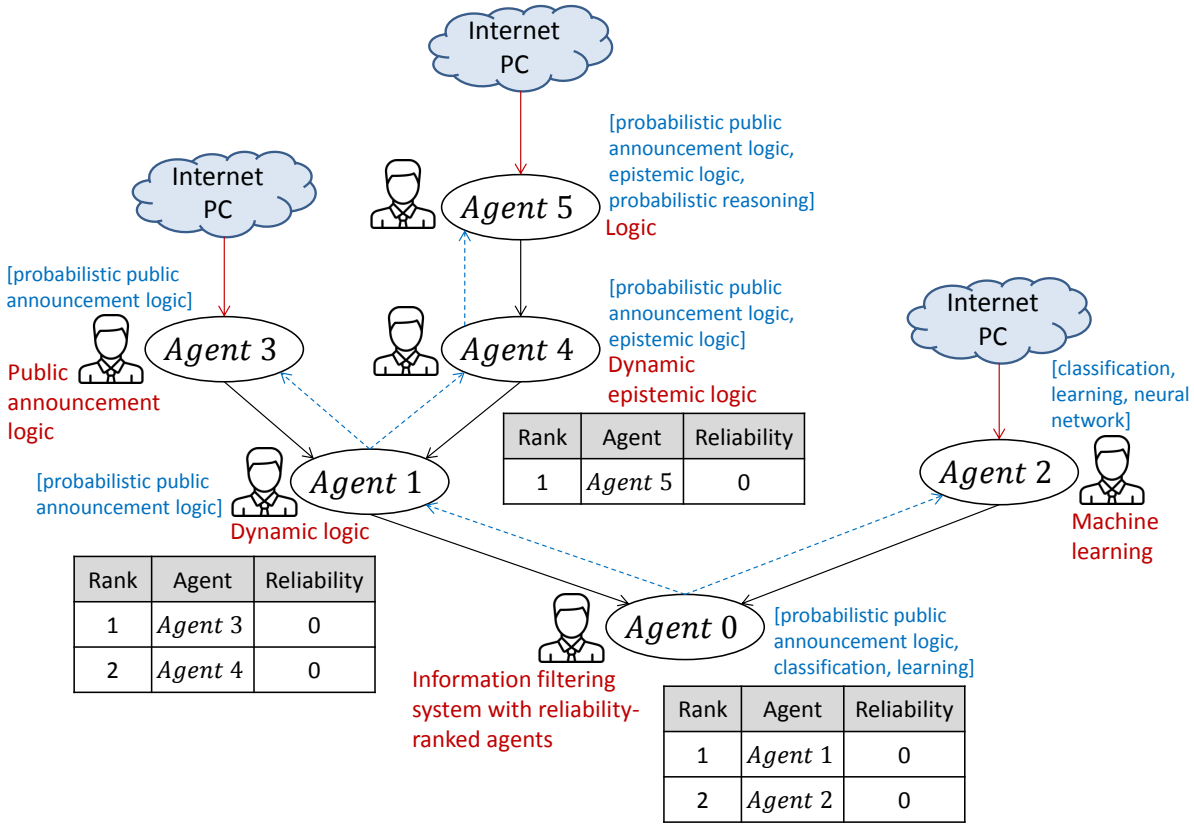


Figure 3.2: An example of the general system model. Six agents specializing in 6 research areas exchange information in a network. Starting from *Agent* 0, a list of keywords is sent upward through all branches to agents in the higher level at which the list of keywords may be preserved or modified so that each receiver agent will keep only keywords that are familiar and meaningful in its research area. The keyword list is propagated until it reaches leaf node agents. Initially, the reliability of higher level agents are set to 0 since no article has been sent. Six humanoid icons represent six researchers who use six agents as support tool in filtering and exchanging articles.

Root node *Agent* 0, as starting point, sends a list of keywords[1] to its child nodes *Agent* 1 and *Agent* 2, asking them to provide interesting articles relating to probabilistic

[1] the list of keywords given by *Agent* 0's user (the researcher who utilizes it)

public announcement logic, classification and learning. After receiving these keywords from *Agent* 0, *Agent* 1 eliminates the two classification and learning that are not typical representative keywords in it research area dynamic logic, while *Agent* 2 removes keywords probabilistic public announcement logic with the same reason and adds an extra keyword neural network into the list. *Agent* 1 then sends the modified list of keywords to the higher level *Agent* 3 and *Agent* 4 whereas *Agent* 2 does not because it is already at a leaf node of the tree-structured network. In the higher layer, *Agent* 3 preserves the list of keywords received from *Agent* 1, *Agent* 4 adds one extra keyword epistemic logic into the list and sends it to *Agent* 5 that in turn adds an additional keyword probabilistic reasoning.

Thus far, each agent has acquired a new list of keywords. Before explaining the role of the new keyword list, let us take an overview of how to train an agent so that it can capture what its owner's interests in his/her research area are. Each agent is provided:

- a default profile of keywords expressing the research area and main research interests,
- and a database of training data that contains a set of articles considered as highly interesting and relating to the research area

of the researcher who uses it. The agent then trains its neural network using the above two items as input (details of training agent's neural network will be discussed later). When the agent obtained a new list of keywords, it may add these keywords to the default profile of keywords if the enlarged profile is consistent with its user's habit. For leaf node agents, the new list of keywords will also be used for searching articles on the Internet.

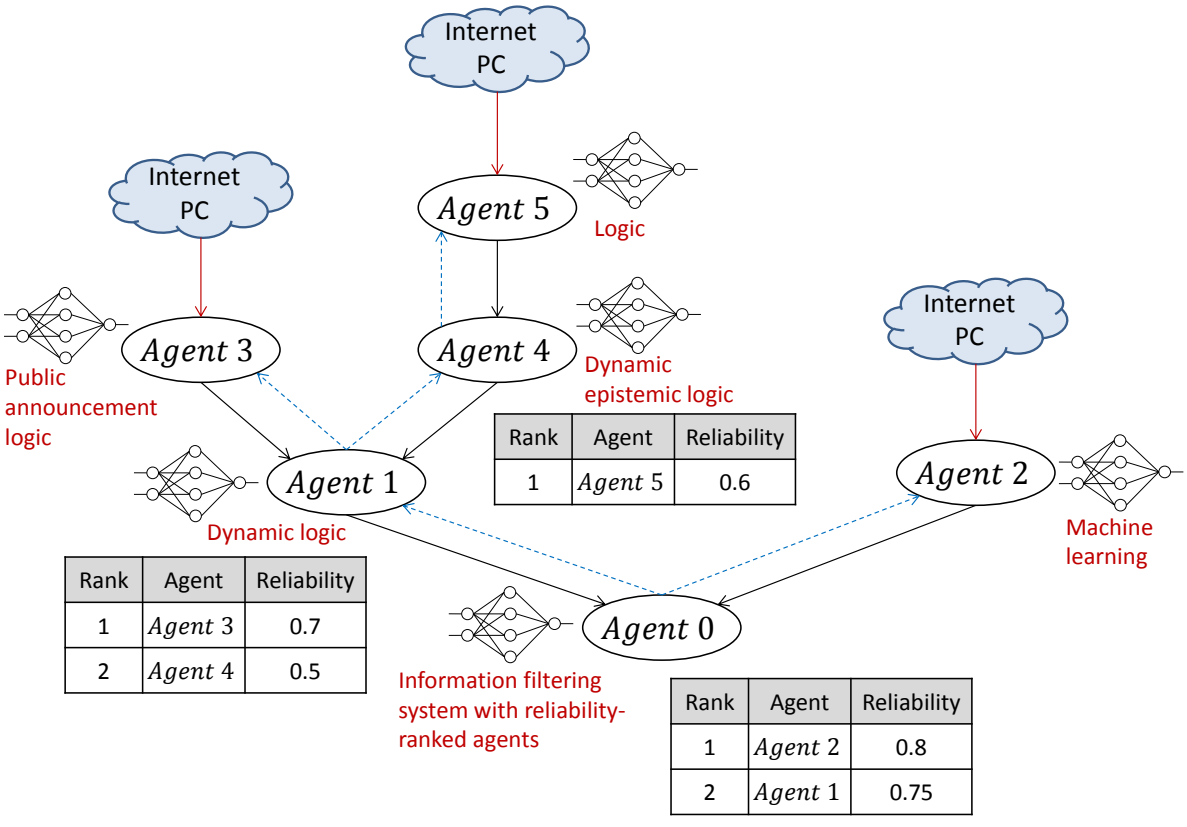


Figure 3.3: An example of the general system model - using neural networks to learn users' research areas and interests. Each agent in the example model (introduced in fig. 3.2) has its own neural network for capturing the research area of its user. Trained neural networks are used to calculate how much a newly loaded article is interesting or related to their users. Reliability that expresses the quality of being trustworthy of sending agents is updated correspondingly.

Figure 3.3 shows agents' utilization of neural networks in learning their users' expertise. Possessing a separate neural network, each agent will independently be able to acquire, update and modify its knowledge about the range of reading interest of its user. For example, *Agent* 5 may add the new list of keywords received from *Agent* 4 (see fig. 3.2) and uses the modified profile of keywords that mainly describes its user's reading interest about logic as well as probabilistic public announcement logic, epistemic logic and probabilistic reasoning to create its new neural network. This new network then learns its user's interest based on the given training article database. While agents are exchanging articles together, the reliability updating tables are updated so that agents that are more effective and accurate will be assigned higher reliability and vice versa. Reliability ranking results will be sent as feedback to successively higher level agents with the purpose of encouraging them to improve their accuracy in selecting and sending articles.

3.3 Scoring Article's Interestingness Using Neural Networks

In this section, we will discuss about two types of neural networks

- Backpropagation Neural Network (BPNN),
- and Self-Organizing Map (SOM)

that are used in this thesis to calculate how interesting articles downloaded from the Internet or loaded from personal computer are. BPNNs are different from SOMs in the perspective that they apply supervised learning paradigm for training step. This thesis however also introduces the SOMs, that are artificial neural networks using unsupervised learning, with the main purpose of demonstrating compatibility of the general model proposed in the previous section - our model of information filtering does not depend on type of neural networks with different learning rules in the training step.

3.3.1 Backpropagation Neural Network Approach

Vectors of Input Data

In construction of a BPNN for capturing a user's research interests, the first step is to design input patterns (we will call them as *vectors of input data* from now on) for the network.

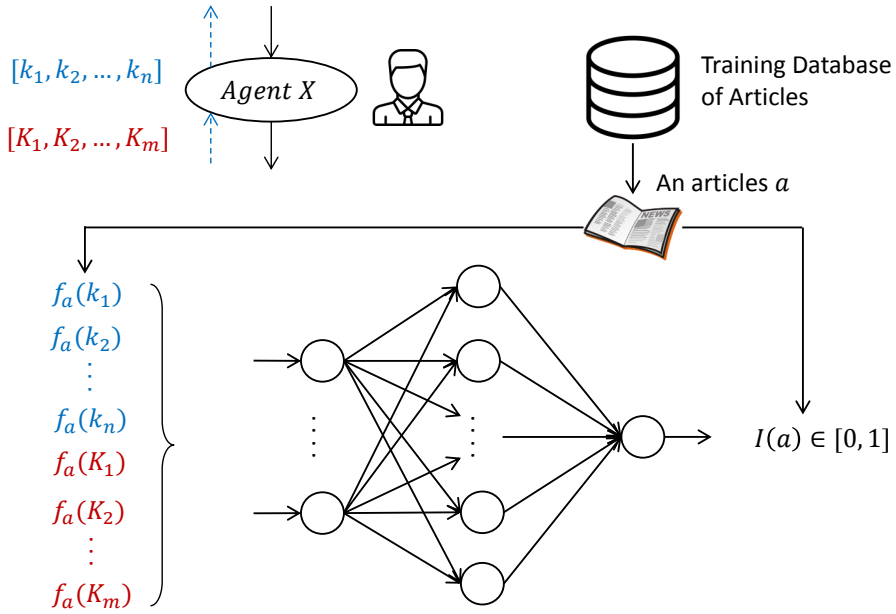


Figure 3.4: Input data for training BPNN. $[k_1, k_2, \ldots, k_n]$ is the list of n keywords received from lower level agent. $[K_1, K_2, \ldots, K_m]$ is the profile of m keywords discribing its user's research area (see fig. 3.2 for a concrete example). $f_a(k_i)$ and $I(a)$ are the frequency of keyword k_i, and a value in the interval $[0, 1]$ that expresses the degree of interestingness of an article a, respectively.

Figure 3.4 shows structure of each vector of input data that is used for training the BPNN of an arbitrary *Agent X* in the agent network. Structure of the vector is strictly related to the structure of the BPNN. In this thesis, each input vector is a one-dimensional array which contains frequencies of keywords in *the list received from other agent and the profile given by the user* $[f_a(k_1), f_a(k_2), \ldots, f_a(k_n), f_a(K_1), f_a(K_2), \ldots, f_a(K_m)]$. The number of neurons in the input layer of the BPNN (details of the BPNN will be discussed

in the next subsections) is equal to the size of the input vector $(n + m)$. For each article a in the training database given by the user, one input vector of keyword frequencies is created and presented to the BPNN. In addition, the article's interestingness $I(a)$ is also necessary during the training step because BPNN uses supervised learning. $I(a)$ is subjectively evaluated by the user based on his/her preferences in reading research papers.

Error-Correction Rules

Given a vector of frequencies $F(a) = [f_a(k_1), f_a(k_2), \ldots, f_a(k_n), f_a(K_1), f_a(K_2), \ldots, f_a(K_m)]$ and $I(a)$ for each article a in the training database, the supervised-learning BPNN will be trained using the *error-correction rules* so that the trained BPNN will return the actual interestingness $I^{'}(a)$ that is approximately equal to the desired one $I(a)$. Essentially, the key principle of error-correction rules during the learning process is to calculate the difference[2] $(I^{'}(a) - I(a))$ to modify the BPNN's internal parameters so that this difference will be gradually reduced. Because this difference describes how biased the actually calculated interestingness $I^{'}(a)$ is in relative to the desired interestingness $I(a)$, reducing it by updating the network's inside parameters means we adjust these parameters to obtain an updated network that returns the interestingness all most equal to what the user predetermined.

Artificial Neurons

Each artificial neural network consists of a set of artificial neurons (or units) whose computational model was inspired by research on human brain's nerve net and neuron.

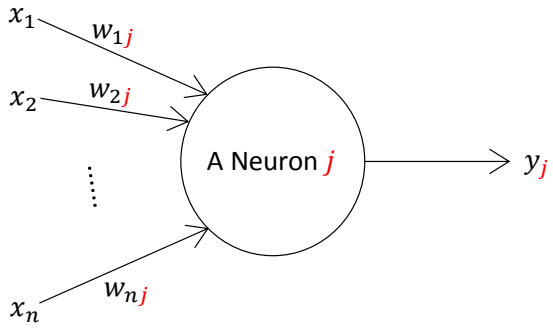


Figure 3.5: A neuron in the BPNN. $x_1, x_2, \ldots, x_n$ are inputs, $w_{1j}, w_{2j}, \ldots, w_{nj}$ are the weights of the input signals and y_j is output of each neuron j.

A typical neuron j, as illustrated in fig. 3.5, has n inputs $x_1, x_2, \ldots, x_n$ and n links connecting from other neurons to it. These links are correspondingly assigned with n weights[3] $w_{1j}, w_{2j}, \ldots, w_{nj} \in \mathbb{R}$. The total input X_j of a unit j is a linear summation function of multiplications between the inputs x_i and the weights wij on these links

$$X_j = \sum_{i=1}^{n} x_i w_{ij} \tag{3.1}$$

Each neuron j has a real valued output y_j. In this thesis, the standard sigmoid function of the total input X_j is chosen to calculate y_j

$$y_j(X_j) = \frac{1}{1 + e^{-X_j}} \tag{3.2}$$

[2] difference between the actually returned output and the desired output is also called the error signal
[3] weights are also used to refer to internal parameters in the Error-Correction Rules

It is not necessary to use exactly the functions given in equations (3.1) and (3.2) for the computational model of sigmoid neurons in BPNN. Linearly combining the inputs to a neuron before using the nonlinear function on its output however significantly reduces the complexity of the learning process [24]. In addition, two main reasons for using the sigmoid function to calculate the output are:

1. The sigmoid function in eq. (3.2) has the property similar to the input-output relationships of biological neurons that our artificial neurons try to stimulate. Specifically, biological neurons map their electric input signals onto $\{0, 1\}$: 0 for the case the voltage exists, 1 for the opposite case. This mapping is normally called the sign function. Since
$$\lim_{X_j \to -\infty} y_j(X_j) = 0, \qquad \lim_{X_j \to +\infty} y_j(X_j) = 1,$$
$y_j(X_j)$ that maps its inputs into the interval $(0, 1)$ can be seen as a *smoothed* version of the sign function [1].

2. The derivative of y_j with respect to X_j can be represented by itself
$$\begin{aligned}
\frac{dy_j}{dX_j} &= \left(\frac{1}{1+e^{-X_j}}\right)' \\
&= \frac{1'(1+e^{-X_j}) - (1+e^{-X_j})'}{(1+e^{-X_j})^2} \\
&= \frac{e^{-X_j}}{(1+e^{-X_j})^2} \\
&= \frac{1+e^{-X_j}}{(1+e^{-X_j})^2} - \frac{1}{(1+e^{-X_j})^2} \\
&= \frac{1}{1+e^{-X_j}} - \frac{1}{(1+e^{-X_j})^2} \\
&= y_j - y_j^2 \\
&= y_j(1-y_j).
\end{aligned} \tag{3.3}$$
Even if the sigmoid function is the general version with a slope degree coefficient β
$$\sigma(x) = \frac{1}{1+e^{-\beta x}},$$
$d\sigma(x)/dx$ can still be represented by just using $\sigma(x)$, namely $-\beta\sigma(x)(1-\sigma(x))$. During the learning process of the BPNN, it is proved that the derivatives dy_j/dX_j are helpful for updating weights assigned to links between neurons [24], and as the above eq. (3.3) shows we now know that dy_j/dX_j can be calculated directly from y_j without requiring extra memories, weights therefore can be adjusted without the necessary of a large amount of memories (note that there may be thousands of weights need to be adjusted for each training pass of the BPNN, that may lead to memory shortage for training process).

Normally, a threshold $\theta_j = x_0$ is added to the weighted sum in eq. (3.1) as a way of shifting the graph of output $y_j(X_j)$ horizontally by θ_j units of length on the two-dimensional Cartesian coordinate plane:
$$X_j = \left(\sum_{i=1}^{n} x_i w_{ij}\right) + \theta_j = \left(\sum_{i=1}^{n} x_i w_{ij}\right) + x_0 w_{0j} = \sum_{i=0}^{n} x_i w_{ij} = \sum_{i} x_i w_{ij}, \tag{3.4}$$

where $w_{0j} = 1$.

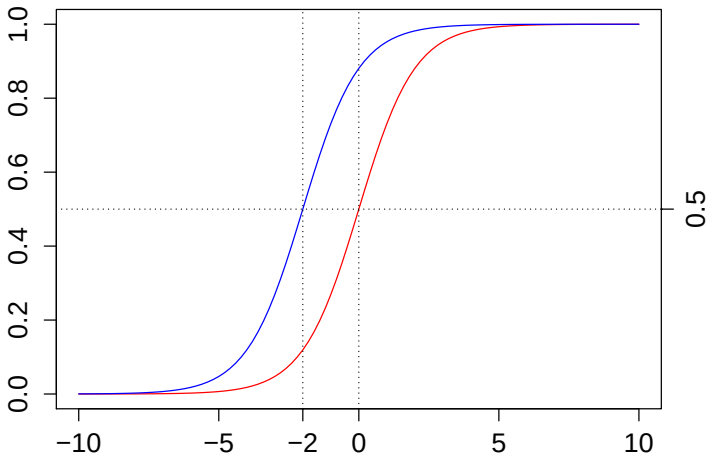


Figure 3.6: Example of sigmoid functions with and without threshold. The red curve is the graph of function $\sigma_1(x) = 1/(1 + e^{-x})$, and the blue curve graphically expresses function $\sigma_2(x) = 1/(1 + e^{-(x+2)})$. Threshold $\theta = 2$ is the units of length we need to shift the red curve left to obtain the blue one.

Adding θ_j to the parameter X_j of the standard sigmoid function (3.2) allows us to adjust the range of X_j in which the function output is greater than or equal to a constant and vice versa. If the linear weighted sum $X_j = \sum_{i=1}^{n} x_i w_{ij}$ is much greater than $-\theta_j$, the output of $y_j(X_j + \theta_j)$ is asymptotic to 1; if X_j is much less than $-\theta_j$, $y_j(X_j + \theta_j)$ will approach to 0; and if it is extremely close to $-\theta_j$, $y_j(X_j + \theta_j)$ is close to 0.5. Figure 3.6 clearly illustrates the range of parameter x is changed by -2 for $\sigma_2(x)$ to attain the same outputs that are equal to $\sigma_1(x)$, namely $\sigma_2(x - 2) = \sigma_1(x)$. Generally speaking, the role of θ_j in the sigmoid function is to tune the neuron's output range dynamically without considering the method used to calculate the total input.

Warren McCulloch and Walter Pitts proposed the first artificial neuron with the *linear threshold computation model* in which the total input is also the weighted sum of the inputs, but the output is calculated by a logic function that returns 0 or 1 and is applied to construct neural networks that did not perform learning process [18, 26]. However, networks of such kind of neurons cannot be trained if they have more than one layer, as needed to tackle even some simple problems [27]. The general classes of problems therefore would probably not be solved by using the computational model introduced by McCulloch and Pitts as they require training multiple-layer networks.

We have specified the detailed structure and computational model for linear sigmoid neurons that we will call them as *linear sigmoid units* (or *units* for short). The sigmoid function and the output are also called *activation function* and *activation value* (or *activation* for short), respectively. In the next subsection details of the BPNN will be explained.

Backpropagation Neural Network (BPNN)

In this subsection, we first describe the general architecture of BPNNs and their learning mechanism. Understanding the general network will help us to gain an accurate and deep intuitive understanding of the BPNN. We then point out how it can be applied for calculating articles' interestingness. All reasons for why the sigmoid function is used as

activation function of each unit will be gradually obvious when the network's learning process is explained.

Three main basic features of a general BPNN are as follows:

- The computational model of each unit in the network includes a nonlinear activation function that is *differentiable*[4].
- The network consists of one input layer, one output layer and one or more layers that are in between both the input and output ones but *hidden* from (invisible to) the outside world.
- Connections of units in between layers are assigned with weights that are all adjusted during the learning process (training step).

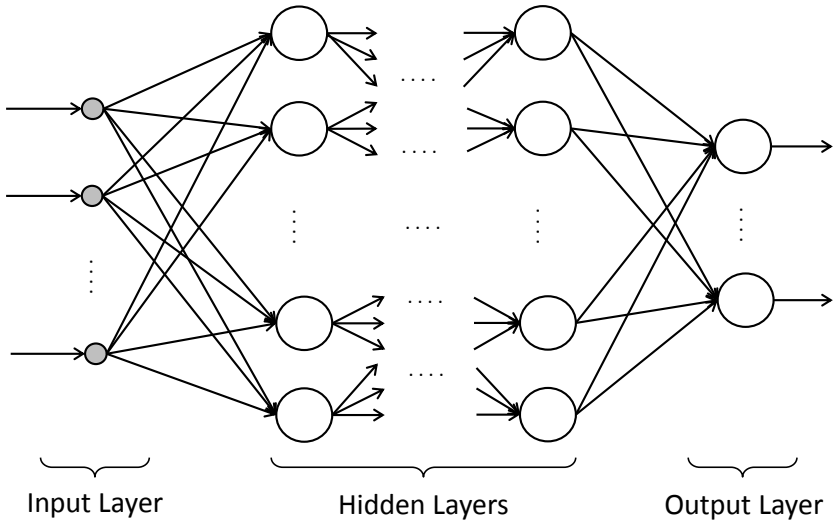


Figure 3.7: General backpropagation neural networks. Each neural network may have one or more hidden layers. Units (neurons) in between layers are fully connected by links showing flow of processing input data that we call it as forward pass, from left to right and layer-by-layer order.

Figure 3.7 depicts the general architecture of BPNNs. Conventionally, each BPNN has (*i*) one input layer in which the number of units is equal to the size of input data vector, (*ii*) one output layer in which the number of units depends on how output data is interpreted, e.g. if the output data is a vector of binary numbers (network is trained so that each unit in the output layer will return values that are approximate to 0 or 1) that encode 8 natural numbers $0, 1, \ldots, 7$, then the output layer will need $\log_2 8 = 3$ units, (*iii*) one or more hidden layers. The number of hidden layers and number of units in each layer will be explained later.

The units in hidden layers play the role of detecting features of inputs, and so are important in operation of the BPNN. As the learning process (training step) progresses across the BPNN, the hidden units gradually *discover* the main features that characterize the training data. These *discoveries* is made by applying a nonlinear transformation (e.g. the sigmoid activation function) on the input data (e.g. the total input) into a new space

[4]A real function is said to be differentiable at a point if its derivative exists at that point. The derivative of a function $f(x)$ with respect to the variable x is defined as

$$f^{'}(x) = \lim_{\alpha \to 0} \frac{f(x+\alpha) - f(x)}{\alpha}.$$

If the above limit exists for all $x \in \mathbb{R}$, then $f(x)$ is said to be differentiable.

where the elements (e.g. output values) in this space are more easily separated into classes than could be in the original space [8].

The main purpose in training the BPNN is to find a set of weights assigned to links in the network so that each time an input vector is presented to the input layer, the output layer will return an output vector that is the same as or sufficiently close to the predetermined vector (the desired vector). The training process that uses the error-correction rule is divided into two phases:

1. *Forward phase*: the total input of each unit is computed based on inputs received from units in the left-hand-side adjacent layers, then the activation value is calculated using the sigmoid function. All activation values of units in the same layer are then simultaneously propagated as inputs to all units in the right-hand-side successive layer. This propagation process continues until it reaches the output layer. In the forward phase, the links' weights are not modified.

2. *Backward phase*: an error signal is first calculated by comparing the output of the network with a desired response.The resulting error signal is then propagated through the network, again layer by layer, but this time the propagation is performed in the backward direction (from the output layer back to the input layer). When an unit receives error signals from all other units in the successive layer, all weights connecting between these units are updated. In this second phase, weight adjustments are the main task.

Suppose the training data set for the learning process is

$$\mathcal{T} = \{(\mathbf{x}_k, \mathbf{d}_k)\}_{k=1}^{K}$$

where

K is the number of pairs of input $\mathbf{x}_k$ and desired output $\mathbf{d}_k$ for training,
$\mathbf{x}_k = (x_{k1}, x_{k2}, \ldots, x_{kM})$ and M is the size of the input layer,
$\mathbf{d}_k = (d_{k1}, d_{k2}, \ldots, d_{kN})$ and N is the size of the output layer.

Forward Phase

In the forward phase, input vectors $\mathbf{x}_k$ are presented to the input layer. Note that units in the input layer are merely set up values received from input vector but do not apply the sigmoid function to calculate activation value. For each unit j in the hidden layer or output layer, it total input X_j is calculated using eq. (3.4)

$$X_j = \sum_i y_i w_{ij}, \tag{3.5}$$

where y_i is the activation (output) of unit i in the immediately preceding layer that directly connects to j. The activation value of j is computed using eq. (3.2).

Backward Phase

Let $\mathbf{y}_k = (y_{k1}, y_{k2}, \ldots, y_{kN})$ be the actual output vector returned by the output layer when the input vector $\mathbf{x}_k$ is presented to the network and the forward phase has finished

propagating activation values until the output layer. The *error signal* at the j^{th} unit in the output layer, that is produced when the network receives $\mathbf{x}_k$ as input, is defined by

$$e_{kj} = d_{kj} - y_{kj} \tag{3.6}$$

For each actual output $\mathbf{y}_k$, the *cumulative error* E_k occurred at all units in the output layer is the sum of their error signals. Furthermore, to evaluate the cumulative error, avoiding positive and negative error signals eliminate each other is important. We do so by defining the cumulative error as the sum of squared error signals

$$E_k = \frac{1}{2}\sum_{j=1}^{N} e_{kj}^2 = \frac{1}{2}\sum_{j=1}^{N}(d_{kj} - y_{kj})^2, \tag{3.7}$$

where the scaling factor $1/2$ is multiplied to simplify mathematical calculations in subsequent analysis that will be clear shortly.

Since the training data set $\mathcal{T}$ includes finite K pairs of input-output vectors, the *average error* E_{av} and the *total error*, E, appear over these training pairs is defined as follows

$$E_{av} = \frac{1}{K}\sum_{k=1}^{K} E_k = \frac{1}{2K}\sum_{k=1}^{K}\sum_{j=1}^{N}(d_{kj} - y_{kj})^2, \tag{3.8}$$

$$E = \sum_{k=1}^{K} E_k = \frac{1}{2}\sum_{k=1}^{K}\sum_{j=1}^{N}(d_{kj} - y_{kj})^2. \tag{3.9}$$

The learning process here uses error-correction rules, thus the main task now is how to minimize the total error E. Naturally, desired outputs d_{kj} are given in $\mathcal{T}$ and actual outputs y_{kj} are functions of adjustable weights of the network's internal links, so what we need to archive is to find a mechanism that allows us to modify these weights systematically so that the total error E will be as minimal as possible. There are two strategies for the weight modification: adjusting weights after each input is fetched to the network that is known as *on-line learning*, and adjusting weights after the whole inputs in $\mathcal{T}$ are passed through the network that is named as *batch learning*.

The on-line learning method changes the weights of the network on an example-by-example basis. Therefore, the cumulative error E_k becomes the function to be minimized. This learning method is

- faster than the batch learning [10],
- able to handle large training data set and redundancy in data and avoids the need to store an accumulated weight change [28].

However, a major concern with on-line learning is that the accuracy of weights after the network is trained is usually lower than batch learning, or in other words, the knowledge inside the trained network is not optimal [10].

The batch learning, as its name suggests, changes the weights after all of K training examples in $\mathcal{T}$ are presented to the network. Thus, the average error E_{av} or the total error E is the function to be minimized during the training step. The main advantages of batch learning are

- parallelization of the learning process,
- accurate estimation of the derivative of the total error function E with respect to the weights. This is important in adjusting weights with minimal biases, hence forming

an optimize knowledge base in the network structure.
The batch learning is appropriate when the training data set $\mathcal{T}$ is sufficiently representative of the knowledge that will be trained to the network [10, 17].

In this thesis, we suppose that each agent is given a training database $\mathcal{T}$ at the beginning and $\mathcal{T}$ contains a large enough number of examples expressing the knowledge that will be trained to the agent's neural network. Therefore, the batch learning method is appropriate for the training step. The remaining of this subsection will concentrate on explaining the mathematical mechanism for changing the network's internal weights based on adjusting the total error E.

A weight correction Δ_{ij} is applied to change the weight w_{ij} assigned to the link connecting from unit i to unit j. If w_{ij} changes, the total error E is also changed. Therefore, in order to minimize E (or in other words, to gradually change E into a smaller value) based on changing weights, its partial derivative[5] with respect to each weight, $\partial E/\partial w_{ij}$, is necessary to be computed.

The *chain rule* in calculus suggests

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ij}} \\ &= \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial X_j} \cdot \frac{\partial X_j}{\partial w_{ij}}\end{aligned} \tag{3.10}$$

can be calculated by computing three partial derivatives $\partial E/\partial y_j$, $\partial y_j/\partial X_j$ and $\partial X_j/\partial wij$. Note that in the above two formulas we omit an index k referring to the k^{th} example of the training data set $\mathcal{T}$.

Starting from each unit j in the output layer, we first calculate $\partial E/\partial y_j$. Differentiating eq. (3.9) without considering the index k gives

$$\begin{aligned}\frac{\partial E}{\partial y_j} &= \frac{1}{2} \cdot 2(d_j - y_j)\frac{\partial (d_j - y_j)}{\partial y_j} \\ &= y_j - d_j.\end{aligned} \tag{3.11}$$

We can compute $\partial y_j/\partial X_j$ using eq. (3.3)

$$\begin{aligned}\frac{\partial y_j}{\partial X_j} &= \frac{dy_j}{X_j} \\ &= y_j(1 - y_j).\end{aligned} \tag{3.12}$$

To compute $\partial X_j/\partial w_{ij}$, we differentiate eq. (3.5) (note that the red index i emphasizes that it is the index over the set of all units that connect to j directly)

$$\begin{aligned}\frac{\partial X_j}{\partial w_{ij}} &= \frac{\partial \left(\sum\limits_{i} y_i w_{ij}\right)}{\partial w_{ij}} \\ &= y_i.\end{aligned} \tag{3.13}$$

[5]partial derivative of a function f of several variables $x_1, x_2, \ldots, x_n$ with respect to variable x_i where $1 \leq i \leq n$ is the derivative of f with respect to x_i in which only x_i is the function's variable and the others are considered as constants. Partial derivative of a function f with respect to variable x is usually denoted by $\partial f/\partial x$.

Substituting (3.11), (3.12) and (3.13) into equation (3.10) we have

$$\frac{\partial E}{\partial w_{ij}} = (y_j - d_j)y_j(1 - y_j)y_i. \tag{3.14}$$

Equation (3.14) allows us to adjust the weights of links from all units i in the last hidden layer to units j in the output layer.

For each unit i in the last hidden layer, we compute $\partial E/\partial y_i$ by taking into account all the links starting from i to all units j in the output layer as follows

$$\begin{aligned}
\frac{\partial E}{\partial y_i} &= \sum_j \left(\frac{\partial E}{\partial y_{ij}}\right) && \text{where } y_{ij} \text{ is the output of } i \text{ and the input of } j \\
&= \sum_j \left(\frac{\partial E}{\partial X_j} \cdot \frac{\partial X_j}{\partial y_{ij}}\right) && \text{by chain rule} \\
&= \sum_j \left(\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial X_j} \cdot \frac{\partial X_j}{\partial y_{ij}}\right) && \text{by chain rule} \\
&= \sum_j \left[\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial X_j} \cdot \frac{\partial \left(\sum_i y_{ij} w_{ij}\right)}{\partial y_{ij}}\right] && \text{where } y_{ij} \text{ and } w_{ij} \text{ are the output and weight from unit } i \text{ in the last hidden layer to unit } j \text{ in the output layer respectively} \\
&= \sum_j \left(\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial X_j} \cdot w_{ij}\right).
\end{aligned} \tag{3.15}$$

Equation (3.15) provides us a straightforward way to calculate $\partial E/\partial y_i$ based on $\partial E/\partial y_j$ in eq. (3.11) and $\partial y_j/\partial X_j$ in eq. (3.12) that have been computed to modify the weights of links between the last hidden layer and the output layer. Since $\partial E/\partial y_i$ is known for every unit i in the last hidden layer, by applying equations (3.10), (3.12) and (3.13) we can again compute $\partial E/\partial w_{hi}$ in which h is the index over units in the layer that is immediately preceding of the last hidden layer. Therefore, this process can be repeated to compute $\partial E/\partial y_\beta$ and $\partial E/\partial w_{\alpha\beta}$, where α and β are units of two consecutively earlier layers.

We have shown that $\partial E/\partial w_{ij}$ can be computed backwards consecutively from the output layer through hidden layers and to the output layer. We now consider how it can be used to update the weight w_{ij} that was assigned to the link connecting from i to j. Since the partial derivative $\partial E/\partial w_{ij}$ expresses the rate at which the value of E changes with respect to the change of w_{ij}, one solution of adjusting w_{ij} is to change it by an amount Δ_{ij} proportional to $\partial E/\partial w_{ij}$

$$\Delta_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{3.16}$$

where η is called the *learning rate parameter* that is set prior to the training step. η can be a fixed constant, or it can decay over time as the learning process proceeds. Adjusting η allows us to scale the amount Δ_{ij} used to change the weight w_{ij}. The negation sign indicates that weight w_{ij} is changed in the direction of decreasing the total error E.

The above technique, that uses equation (3.16) - a partial derivative of error with respect to weight - to adjust weights and thereby gradually minimizes error, is called

gradient descent[6]. Gradient descent has been studied and widely applied to adjust weights and optimize error in neural networks. Its advantages are simple implementation and fast for problems whose the training data set is large enough [3, 15].

Although eq. (3.16) provides an approximation mechanism for the weight adjustments, an other critical question indeed needs to be answered. It is obvious that if we set up the learning rate parameter η smaller, the changes to the weights in the network will be smaller from one training pass to the next. This however leads to a lower rate of learning (learning process will last longer). On the other hand, the larger we make the learning rate parameter η in order to speed up the rate of learning, the more unstable the network with the weights obtained may become. A popular method of increasing the rate of learning while avoiding the undesirable instability is to modify eq. (3.16) by introducing a *momentum* term $\alpha \in [0, 1]$, as shown follows

$$\Delta_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}}(t) + \alpha \Delta_{ij}(t-1) \tag{3.17}$$

where the index t at the initial time is set up to 0 and is incremented by 1 for each training pass through the whole examples in the training data set $\mathcal{T}$.

We now have the following formula for adjusting an arbitrary weight w_{ij} of the network

$$\begin{aligned} w_{ij}(t+1) &= w_{ij}(t) + \Delta_{ij}(t) \\ &= w_{ij}(t) - \eta \frac{\partial E}{\partial w_{ij}}(t) + \alpha \Delta_{ij}(t-1) \end{aligned} \tag{3.18}$$

Number of Hidden Layers

Brightwell et al. [4] proved that if networks are trained to map from continuous less-than-or-equal-to-two-dimension input examples onto the set $\{0, 1\}$, that is normally used in two-label classification problems, then one-hidden-layer networks are satisfiable to solve the problems. In general, theoretical researchers have proved that one hidden layer is sufficient for a BPNN to approximate any continuous mapping from the input examples to the outputs with an arbitrary degree of accuracy [2, 7, 21]. Multiple-hidden-layer BPNNs are appropriate in solving problems in which the inputs are vectors of multiple-dimentional vectors. In this thesis, the BPNNs we use to score articles' interestingness receive *input vectors whose each element is an integer showing frequency of a keyword*. The BPNNs reported here, therefore, are designed as single-hidden-layer neural networks. The number of units in the hidden layer is selected as double of the number of unit in the input layer.

Initialization of Weights

Usually the initial weights are randomly selected from the range $[-a, a]$. In literature there are many suggestions how to estimate the value of parameter a. For instance, Nguyen and Widraw [22] proposed the evaluation

$$a = (H)^{\frac{1}{N}}$$

[6]gradient descent is a technique allows us to find a local minimum of a function $f(x)$ by initially choosing any starting point $x = x_0$; then moving to a neighboring point that is downhill, or in other words moving x to the direction that the gradient (derivative) $f^{`}(x)$ is negative, until a local valley is reached, or in other word x approaches to the nearest point from x_0 where $f(x)$ is locally minimal.

for the weights in between the input and hidden layers, where H is the number of units in the hidden layer, N is the number of units in the input layer. For weights in between the hidden and output layers, $a = 0.5$. This method is however applied when the activation function is the tangent function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{3.19}$$

In this thesis, the logistic function is used as activation function, and we choose $a = 0.5$ for both the input, hidden and output layers. All the weights and thresholds are randomly selected from the interval $[-0.5, 0.5]$.

Stopping Criteria

The learning process iterates until a *stopping criterion* is satisfied. Various criteria can be found in literature. In practice, the following three criteria are widely and commonly used to terminate the weight adjustments

1. *Maximum training time:* specifying a training time limit. During the training step, the training algorithm will check whether its running time so far exceeds the maximum limit of allowed time. The training step may go a bit beyond the specified limit in order to complete the final data pass[7].

2. *Maximum training data passes:* specifying a maximum number of training passes $t*$. Training will be stopped when its total training passes t exceeds the maximum: $t > t*$.

3. *Minimum error accuracy:* learning process will continue until the total error E or the average error E_{av} is less than a sufficiently small gradient threshold ε.

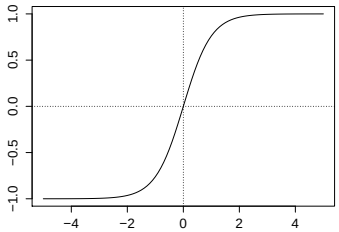
The third criterion may never be satisfied because the gradient descent learning method does not guarantee that local minimum of the total error will always be found by adjusting the network's weights. The research reported in this thesis, therefore, uses both the second and the third criteria specifying both the maximum training data passes $t*$ and the minimum error threshold ε to check the stopping condition of the training step. Training will stop when $E(t) < \varepsilon$ or $t > t*$.

Input Normalization

Normalizing input vectors before presenting it to the input layer is crucial in the succeed of training the network. LeCun et al. in [16] pointed out that in BPNNs using the tangent function[8], input variable should be preprocessed so that its mean value, averaged over the entire training examples, is close to 0, or else it will be small compared to its standard

[7] each data pass is also called an epoch presentation of an input example and get backward phase finished in on-line learning neural networks, or an epoch presentation of all input examples in the training data set and get their backward phases finished in batch learning neural networks.

[8]



the tangent function as defined in eq. (3.19) (graph is on the left hand side) is a sigmoid function whose value varies in $[-1, 1]$ and symmetric through the original coordinate $(0, 0)$. This is one of the main reasons for LeCun et al. in [16] argued that input variable should be normalized so that its mean value over the whole training data set is close to 0, because $\tanh(x)$ only changes if x is around 0 and is asymptotic to 1 or -1 if $x \to \pm\infty$.

deviation[9]. However, the logistic function $f(x) = 1/(1 + e^{-x})$ used in this research has values varying in the interval $[0, 1]$ and symmetric at point $(0, 0.5)$, so input vectors are normalized into the same size vectors that contain elements in the interval $[0, 1]$ and these elements' mean value is close to 0.5 compared to the non-normalized elements' mean value, e.g. the average of $0.2, 0.3$ and 0.9 is closer to 0.5 than the average of $20, 30$ and 90, (frequencies of three keywords).

Another important feature we want to characterize when normalizing input vectors is that elements that are frequencies of keywords expressing the main research interests of the user should be considered as having higher contributions to the output value since we design the higher output value the network returns the more interesting the input article is. We do it by defining a threshold function θ mapping from a set of keywords onto a set of thresholds showing the maximum frequencies of these keywords.

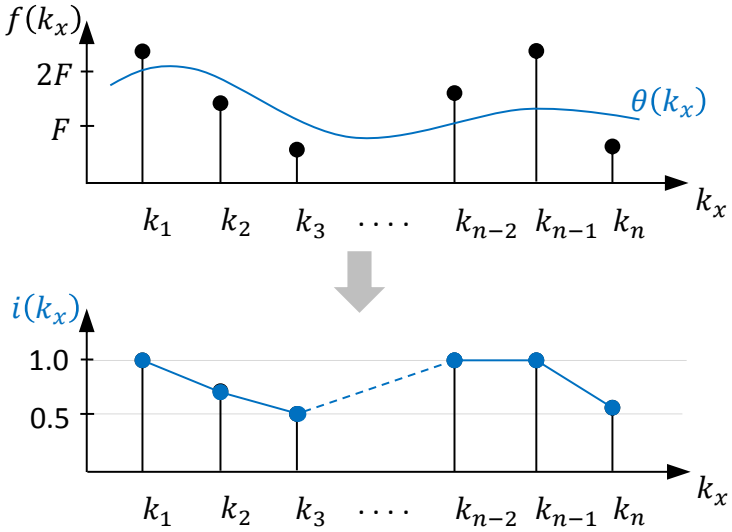


Figure 3.8: Normalizing input vectors. $k_1, k_2, \ldots, k_n$ are keywords whose frequencies are elements of each input vector. $f(k_x)$ is the frequency of keyword k_x. $\theta(k_x)$ is a function defining criterion for evaluating k_x's contribution to the total interestingness of each article, e.g. the allowed maximum frequencies. F and $2F$ are one and two units of frequencies with respect to function θ. $i(k_x)$ is the normalized value of the frequency of keyword k_x, and expresses the contribution of this keyword to the total interestingness of the article characterized by the input vector.

Figure 3.8 depicts the method of normalizing each input vector

$$[f(k_1), f(k_2), \ldots, f(k_n)]$$

into

$$[i(k_1), i(k_2), \ldots, i(k_n)]$$

by defining a threshold function $\theta(k_x)$ that specifies how much the keyword k_x can contribute to the total interestingness of each article. θ is normally defined by users based on their research area and interests.

[9]standard deviation is a measure that is used to quantify the amount of variation or dispersion of a set of data values. A low standard deviation indicates that the data points tend to be close to the mean value of the set, while a high standard deviation indicates that the data points are spread out over a wider range of values.

The definition of function $i(k_x)$ is as follows

$$i(k_x) = \begin{cases} 1, & \text{if } f(k_x) \geq \theta(k_x) \\ \dfrac{f(k_x)}{\theta(k_x)}, & \text{if } f(k_x) < \theta(k_x) \end{cases}. \tag{3.20}$$

Extensibility: The method of normalization described here can be applied to normalize and reduce dimension of input vectors. For instance, if our neural network is used to measure not only the interestingness but also the usefulness of articles, each input vector may be in the form of $[f(k_1), f(k_2), \ldots, f(k_n)]$ with two threshold functions $\theta_i(k_x)$ and $\theta_u(k_x)$ describing the range that each keyword k_x can contribute to the total interestingness and usefulness, respectively.

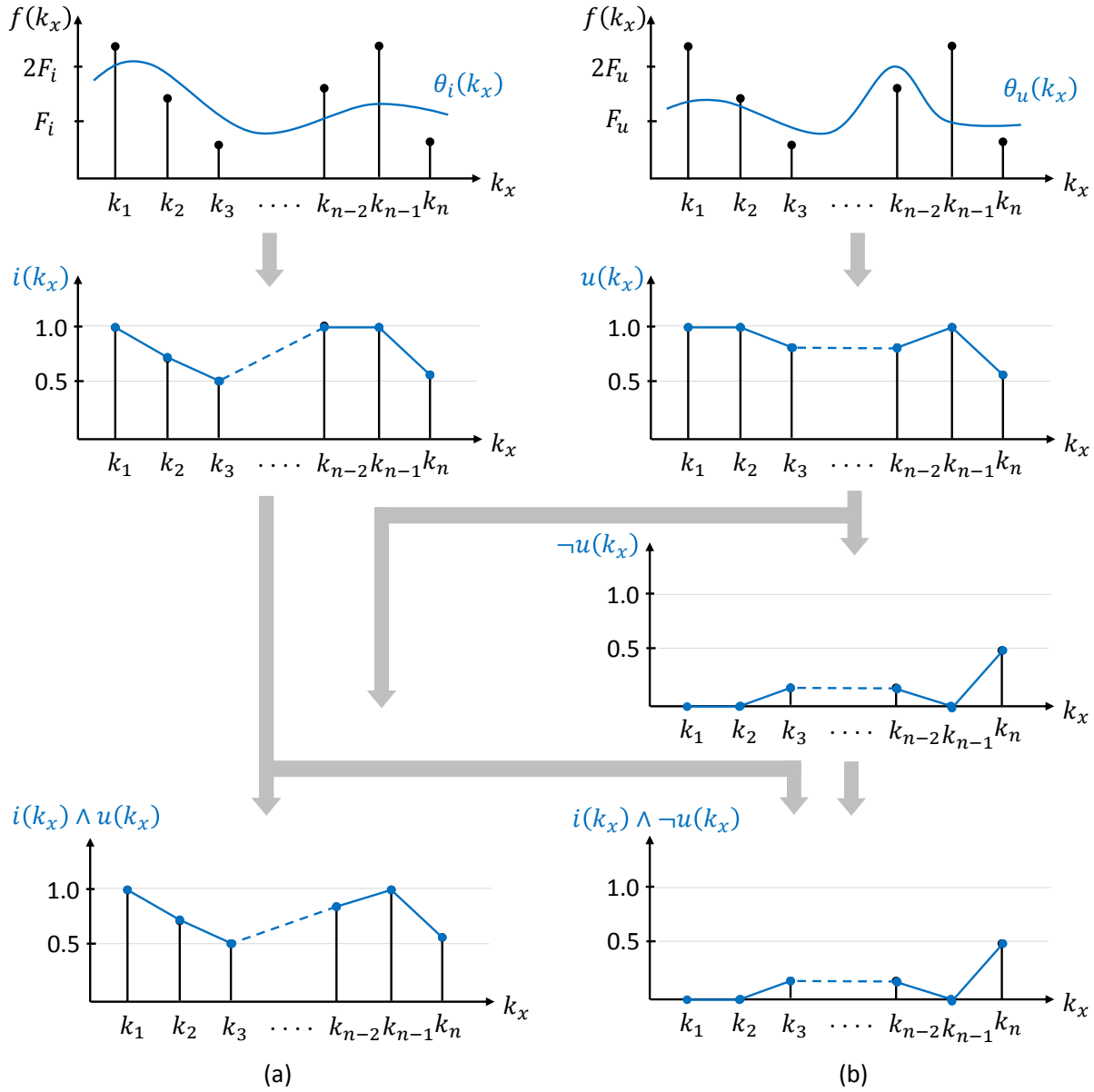


Figure 3.9: Normalizing inputs of multiple criteria. **(a)** shows graph of a normalized input vector that describes the *interesting and useful* degree that each keyword k_x contributes to the total. **(b)** shows graph of a normalized input vector in the case the *interesting but not useful* degree that each keywords contribute to the total.

Using eq. (3.20), each input vector will be normalized into two components that are

represented by two functions $i(k_x)$ and $u(k_x)$ as in fig. 3.9. If our neural network is used to calculate *how interesting and useful* each article is, we then can combine these two components in to one vector whose the x^{th} element is $min(i(k_x), u(k_x))$ as illustrated in graph (a) of fig. 3.9. In case our neural network is used to compute *how much interesting but not useful each article is*, the two components can be combined into one vector in which its x^{th} element has value of $min(i(k_x), \neg u(k_x))$ where $\neg u(k_x) = 1 - u(k_x)$ as illustrated in graph (b) of fig. 3.9. In general, we have the following rules for combining multiple input components together

$$\begin{aligned}
&\text{Negation: } \neg c(k_x) := 1 - c(k_x) \\
&\text{Disjunctions: } \bigvee_{\alpha=1}^{n} c_\alpha(k_x) := \max\{c_\alpha(k_x) \mid \alpha = 1, 2, \ldots, n\} \\
&\text{Conjunctions: } \bigwedge_{\alpha=1}^{n} c_\alpha(k_x) := \min\{c_\alpha(k_x) \mid \alpha = 1, 2, \ldots, n\},
\end{aligned}$$

where $c(k_x), c_\alpha(k_x)$ are components normalized by using eq. (3.20), α is an index over the set of n components being combined.

Summary and Algorithm

We have discussed all necessary details about the BPNN that is used in this research in order to measure the interestingness of research articles. In this subsection, we put everything together, show the final design architecture, and summarize the algorithm for training the neural network.

As discussed in the previous subsection, the BPNN reported in this thesis consists of one input layer, one hidden layer which has a number of hidden unit as twice as the number of input units. The output layer is designed to contain only one unit. This output unit returns activation values in the interval $[0, 1]$ as to indicate the degree of interestingness of articles. Figure 3.10 on page 26 depicts all of these details. The input layer has $(n+m)$ normal units that only perform setting $(n+m)$ input values receive from each input vector

$$[i_a(k_1), i_a(k_2), \ldots, i_a(k_n), i_a(K_1), i_a(K_2), \ldots, i_a(K_m)]$$

normalized from

$$[f_a(k_1), f_a(k_2), \ldots, f_a(k_n), f_a(K_1), f_a(K_2), \ldots, f_a(K_m)]$$

by using eq. (3.20) (see fig. 3.4 for the reason why each input vector has size of $(n+m)$). These $(n+m)$ input units do not use the logistic sigmoid function as defined in eq. (3.2). They only send input values to units in the hidden layer. All $2(n+m)$ units in the hidden layer and one unit in the output layer, however, use the logistic function to calculate their output values. The total error

$$E = \frac{1}{2} \sum_{a=1}^{K} [I(a) - I^{'}(a)]^2,$$

where a is an index over the set of K training articles, will be used to check the whether the network has been successfully trained and the learning process should be stopped. The training algorithm for the BPNN used in this research is on page 27.

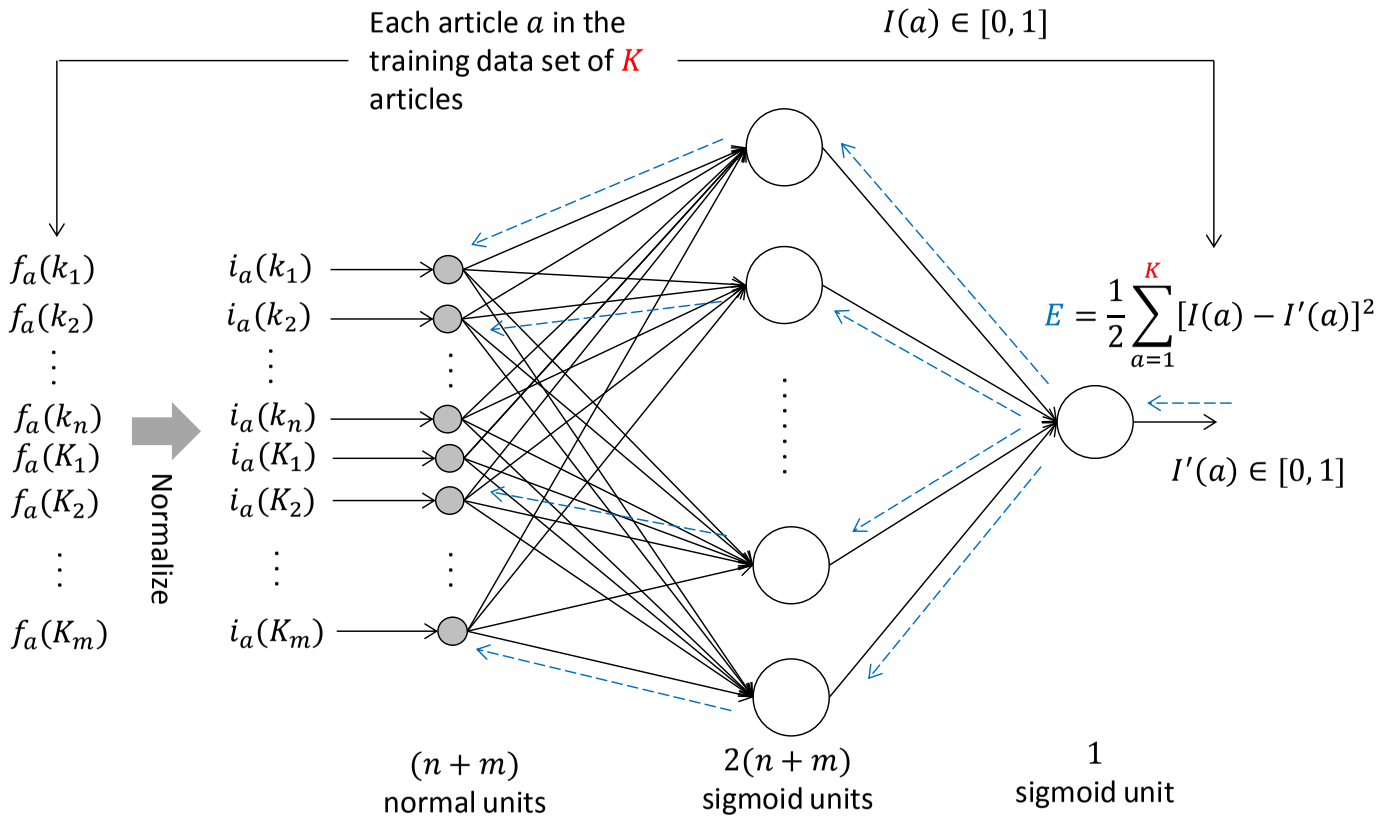


Figure 3.10: Detailed architecture of the BPNN for scoring articles' interestingness. Blue dashed arrows show the directions of backpropagate error for adjusting weights.

Algorithm 1: Training the BPNN for Scoring Articles' Interestingness

Input : Untrained BPNN,
Training data set $\mathcal{T} = \{(F(a), I(a))\}_{a=1}^{K}$ contains K articles a,
Maximum training passes $t*$, maximum error ε,
Learning rate η, and momentum α

Output: Trained BPNN

1: *Initialization:*

- initialize the weights for links in between the input, hidden and output layers randomly in the interval $[-0.5, 0.5]$.
- initialize the total error $E = 0$, the training pass $t = 1$

2: *Forward Phase:* for each training article a whose frequency input vector and interestingness are given as the following pair

$$(F(a) = [f_a(k_1), f_a(k_2), \ldots, f_a(k_n), f_a(K_1), f_a(K_2), \ldots, f_a(K_m)], I(a))$$

- normalize input $F(a)$ into
$$i(a) = [i_a(k_1), i_a(k_2), \ldots, i_a(k_n), i_a(K_1), i_a(K_2), \ldots, i_a(K_m)]$$
and present it to the input layer.
- each input unit receives and forwards the corresponding value from $i(a)$ to all units in the hidden layer.
- each hidden unit computes total input using eq. (3.5) and activation value using eq. (3.2), then forwards the activation value to the output unit. The output unit in term performs the same task of calculating total input and activation value $I^{'}(a)$.
- calculate total error $E = E + \frac{1}{2}[I(a) - I^{'}(a)]^2$

3: *Backward Phase:*

- compute partial derivatives $\partial E/\partial w_{ij}$ for the weights w_{ij} of links between hidden units and the output unit using equations (3.10), (3.11), (3.12) and (3.13) in which the activation value of the output unit j is $y_j = I^{'}(a)$.
- update weights w_{ij} of links between the hidden units and the output unit using eq. (3.18) shown again as follows
$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial E}{\partial w_{ij}}(t) + \alpha \Delta_{ij}(t-1).$$
- compute partial derivatives $\partial E/\partial w_{ij}$ for the weights w_{ij} of links between the input units and the hidden units using equations (3.10), (3.15), (3.12) and (3.13).
- update weights w_{ij} of links between the input units and the hidden units using eq. (3.18).

4: *Stopping Criteria Test:* if the average error $E_{av} = E/K < \varepsilon$ or the number of training passes $t > t*$, then stop the training; else set up $E = 0$, increase $t = t + 1$, go back to step 2 *Backward Phase* and repeat for the next training pass until one of the stopping criteria is satisfied.

3.3.2 Self-Organizing Map Approach

In this section, an other type of neural network, namely the self-organizing map (SOM), will be explained and shown how it can be applied to measure the interestingness of articles. Recall that the purpose of introducing both BPNN and SOM is to emphasize the flexibility of our framework of filtering information. In particular, we want to demonstrate that both supervised and unsupervised learnings can be used to scoring articles' interestingness, so there may be a variety of other methods that can play the same role of the BPNN and SOM applied in this research.

General Model

The main characteristic that differentiates SOMs from other types of neural networks is that they map nearby input patterns onto nearby output units (output neurons) on the map. This ideal was inspired from biological research on the cortex of highly developed animal brains [9]. Figure 3.11 depicts the essential architecture of a SOM in which the map is a *four-by-four* lattice of output units. Each output unit is fully connected from all units in the input layer via links that are assigned with weights. In order to keep the picture easier to understand, only some links are drawn. The weights of all links connecting to an output unit form a weight vector for the unit.

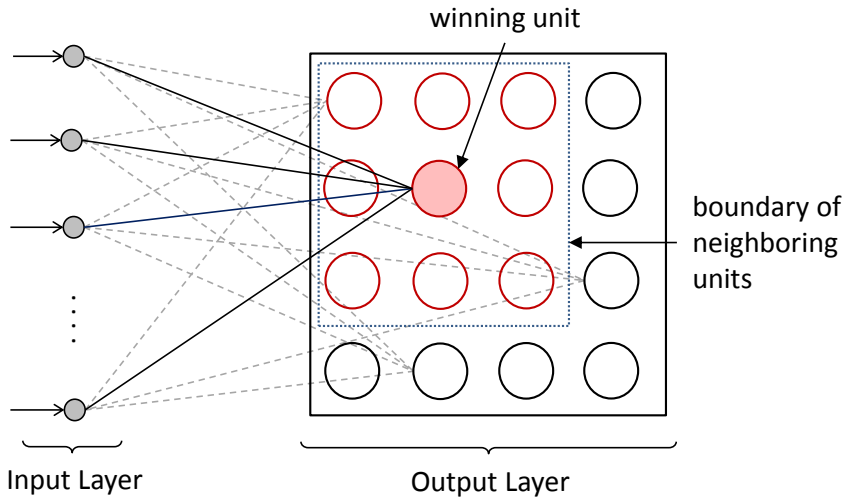


Figure 3.11: General architecture of two-dimensional self-organizing map. Each output unit is connected with all input units. The winning unit is the one whose weight vector is best matching with the input pattern compared to the other ones. All units around the winning unit but not go beyond the boundary are called neighboring units. The boundary of neighborhoods needs not to be square, but may also be rectangular, hexagonal, etc.

Generally, each SOM has m output units, arranged in a one-, two- or more-than-two-dimensional lattice, and n input units receiving n-dimensional input vectors (patterns). However, the two-dimensional SOMs are usually already effective enough for approximating similarity relations of high-dimensional data items [13]. We therefore will concentrate on these two-dimensional maps in this thesis. Let an input vector randomly selected from the input space be denoted by

$$\mathbf{x} = [x_1, x_2, \ldots, x_n]. \tag{3.21}$$

Since there are m output units, there are accordingly m weight vectors associated with them. Let $\mathbf{w}_j$ denote one of these weight vectors

$$\mathbf{w}_j = [w_{j1}, w_{j2}, \ldots, w_{jn}], \qquad 1 \leq j \leq m. \tag{3.22}$$

SOMs are designed to classify a set of continuous-valued input vectors $\mathbf{x}$ into m or less-than-m clusters using self-organization process. During the self-organization process, the c^{th} output unit whose weight vector $\mathbf{w}_c$ most closely matches with the input pattern $\mathbf{x}$ is chosen as the winning unit. Typically, the minimum of the Euclidean distance[10] between the input pattern and the weight vector is the criterion for searching the winning unit

$$c = \underset{j}{\operatorname{argmin}}\{\|\mathbf{x} - \mathbf{w}_j\|\}. \tag{3.23}$$

After the winning unit c is identified, its neighboring units will next be determined by a *neighborhood function*

$$N : \text{index } k \text{ of a unit} \rightarrow \text{a set of indices of the unit } k\text{'s neighboring units.} \tag{3.24}$$

Note that, we read the index of a unit as its name, i.e. unit k is the k^{th} unit. In general, the weight vectors of neighboring units are not close to the input pattern. All the weight vectors associated with the winning unit c and its neighboring units $N(c)$ are updated in a direction that the modified weight vectors will match better with the input. The following formula is commonly used for update

$$\mathbf{w}_j = \mathbf{w}_j + h_{cj}[\mathbf{x} - \mathbf{w}_j], \qquad j \in \{c\} \cup N(c), \tag{3.25}$$

where h_{cj} is called the *weight adjusting factor function*[11] showing the rate of the weight modifications. The self-organizing algorithm trains the map with assumption of the above equation will converge and produce the wanted order for weight vectors and so the order for output units will be. Rewriting eq. (3.25) in the form of coordinates we have the following formula for adjusting weights assigned to links connecting from all input units to output units $j \in \{c\} \cup N(c)$ illustrated as the dark red area in fig. 3.11

$$\begin{aligned}[w_{j1}, w_{j2}, \ldots, w_{jn}] = \\ [w_{j1} + h_{cj}(x_1 - w_{j1}), w_{j2} + h_{cj}(x_2 - w_{j2}), \ldots, w_{jn} + h_{cj}(x_n - w_{jn})].\end{aligned} \tag{3.26}$$

[10] the Euclidean distance between two vectors $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and $\mathbf{y} = [y_1, y_2, \ldots, y_n]$ in Euclidean n-space is defined by the Pythagorean formula

$$\|\mathbf{x} - \mathbf{y}\| = \|\mathbf{y} - \mathbf{x}\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

The smaller the Euclidean distance between two vectors is, the more similar they are. Sometimes, in order to avoid having to calculate the square root, the square of Euclidean distance is used to measure the similarity between two vectors.

[11] in many literature, the *weight adjusting factor function* is called as the *neighborhood function*. In this thesis, however, in order to make things distinguished we separate the neighborhood function N, whose task is to determine neighboring units, from the weight adjusting factor function h_{cj} whose main task is to determine the rate of adjusting the weights associated with the winning unit c and its neighboring units $j \in N(c)$.

Each time eq. (3.25) is used to update weight vectors associated with units in the set $\{c\} \cup N(c)$, the map's state is changed. We trace the map's states by introducing a parameter t, showing the number of iterations in weight updates, into equations (3.23), (3.24) and (3.25) as follows

$$c = \underset{j}{\operatorname{argmin}}\{\|\mathbf{x}(t) - \mathbf{w}_j(t)\|\}, \tag{3.27}$$

$$N(k)(t) : (\text{unit } k, \text{ time } t) \rightarrow \text{a set of indices of the unit } k\text{'s neighboring units at time } t, \tag{3.28}$$

$$\mathbf{w}_j(t) = \mathbf{w}_j(t) + h_{cj}(t)[\mathbf{x}(t) - \mathbf{w}_j(t)], \qquad j \in \{c\} \cup N(c)(t). \tag{3.29}$$

The extremely popular choice of the weight adjusting factor function $h_{cj}(t)$ is

$$h_{cj}(t) = \alpha(t) \exp\left(-\frac{\|c - j\|^2}{2\sigma^2(t)}\right) \tag{3.30}$$

where $\alpha(t)$ is the *learning rate* that is a slowly decreasing function of time, $\|c - j\|$ is the Euclidean distance between the winning unit c and its neighboring unit j, and $\sigma(t)$ is another decreasing function that is the half of the square neighboring area's edge or the radius of the circular neighboring area. $\sigma(t)$ helps the function $N(c)(t)$ to determine all neighboring units of the winning unit c at time t. Figure 3.12 illustrates the case in which the neighboring area is a square with edge of $2\sigma(t_0)$ at the beginning (equal to the map's size). Its size, during the training step is gradually reduced to $2\sigma(t)$ at time t, and may be decreased to contain only the winning unit.

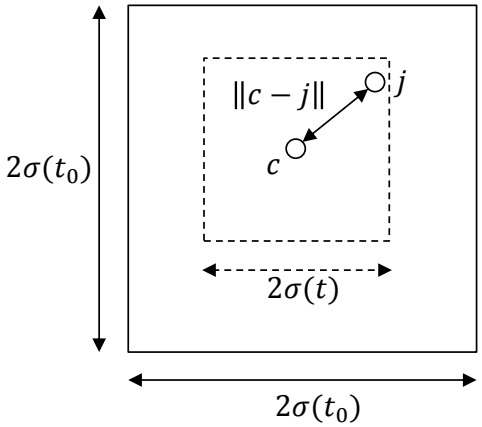


Figure 3.12: Decrease in the size of the neighboring area on SOM. Initially, the size of the neighboring area may be equal to the size of the map and equal to $2\sigma(t_0)$. Function σ is then gradually decreased to $2\sigma(t)$ at time t during the training that makes the number of neighboring units of the winning unit is reduced over time. At the end, the area is small even may be only the winning unit, and map obtains its stable state.

Since $2\sigma^2(t)$ is the square of the one half of the diagonal of the neighboring area and c is in the center of the area, we always have

$$0 \leq \|c - j\|^2 \leq 2\sigma^2(t)$$
$$\Rightarrow 0 \geq -\frac{\|c - j\|^2}{2\sigma^2(t)} \geq -1.$$

At the beginning of the training, $\sigma(t) = \sigma(t_0)$. The further the training progresses, the smaller $\sigma(t) < \sigma(t_0)$ will become. This leads to

$$-\frac{\|c-j\|^2}{2\sigma^2(t)} \text{ tends to reduce from 0 to } -1.$$

Therefore,

$$\exp\left(-\frac{\|c-j\|^2}{2\sigma^2(t)}\right) \text{ tends to reduce from } \exp(0) = 1 \text{ to } \exp(-1) \approx 0.3679$$

as illustrated in fig. 3.13. $\exp(-\|c-j\|^2/2\sigma^2(t))$ is also called a *Gaussian function*.

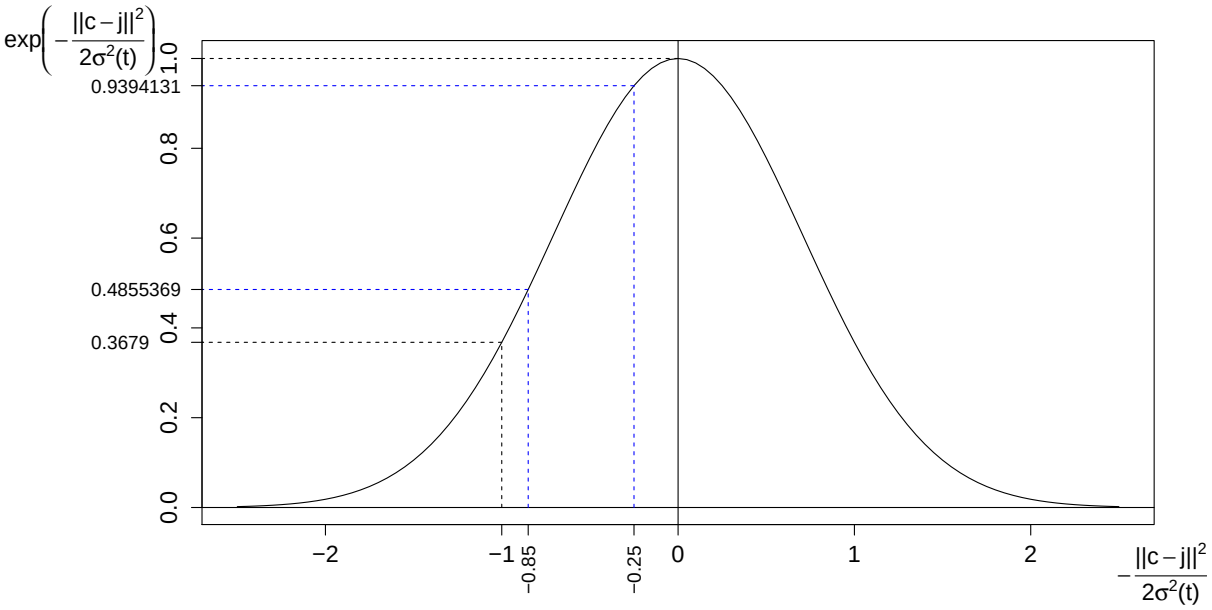


Figure 3.13: Gaussian function for adjusting weights. During the learning process, t increases from t_0 to the current time (current iteration) t, σ reduces from $\sigma(t_0)$ to $\sigma(t)$, $-\frac{\|c-j\|^2}{2\sigma^2(t)}$ reduces from 0 to -1, and $\exp\left(-\frac{\|c-j\|^2}{2\sigma^2(t)}\right)$ reduces from 1 to 0.3679.

The learning rate $\alpha(t)$ should also be a decreasing function over time t so that the multiplication of $\alpha(t)$ and function $\exp(-\|c-j\|^2/2\sigma^2(t))$ will create a decreasing function $h_{cj}(t)$ of increasing time t. There are various functions satisfying to be selected as $\alpha(t)$, i.e.

Inverse of time: $\alpha(t) = \frac{1}{t}$;

Linear of time: $\alpha(t) = 1 - \frac{t}{t_{\max}}$, where $t_{\max}$ is the maximum of t;

Heuristic of time: $\alpha(t) = \alpha(t_0)\exp\left(-\frac{t}{T}\right)$, where t_0 is the initial time and T is a time constant and should be greater than $t_{\max}$.

In this thesis, $\alpha(t)$ is defined as follows

$$\alpha(t) = \alpha(t_0)\left(\frac{\alpha(t_{\max})}{\alpha(t_0)}\right)^{\frac{t}{t_{\max}}}, \tag{3.31}$$

where $\alpha(t_0) = 1.0$, $\alpha(t_{\max}) = 0.005$. Substituting real values into eq. (3.31) gives

$$\alpha(t) = (0.005)^{\frac{t}{t_{\max}}} \tag{3.32}$$

Figure 3.14 illustrates two learning rate functions $\alpha(t)$, one with $t_{\max} = 20$, and the other with $t_{\max} = 1000$. It is easy to realize, from these two functions' graphs, that the learning rates $\alpha(t)$ monotonically decrease with strikingly similar ratio when t increases in both cases although 20 is quite different from 1000.

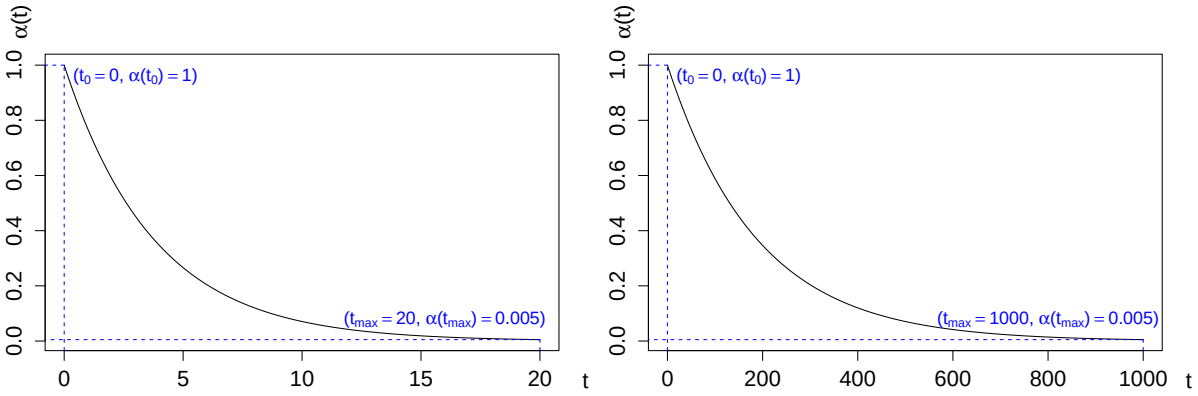


Figure 3.14: Learning rate function for SOM. $\alpha(t)$ monotonically decays with the very similar graph in both cases $t_{\max} = 20$ and $t_{\max} = 1000$. Furthermore, both learning rates are gradually asymptotic to $\alpha(t_{\max})$ and become stable when $t \to t_{\max}$.

The decreases of $\alpha(t)$ and $\exp(-\|c - j\|^2/2\sigma^2(t))$ combine together making $h_{cj}(t)$ reduced over time t, thus the amounts $h_{cj}(t)[\mathbf{x}(t) - \mathbf{w}_j(t)]$ that are used to adjust weight vectors $\mathbf{w}_j(t)$ are gradually smaller with increasing t. Note that, while the Gaussian function reduces to shrink the winning area (to reduce the number of output units whose associated weights need to be modified), the learning rate function reduces with the purpose of decreasing the amounts of weight changes.

Using the similar strategy of decreasing learning rate $\alpha(t)$ as defined in eq. (3.31), in this thesis the $\sigma(t)$ function is defined as follows

$$\sigma(t) = \sigma(t_0)\left(\frac{\sigma(t_{\max})}{\sigma(t_0)}\right)^{\frac{t}{t_{\max}}}, \tag{3.33}$$

where $\sigma(t_0) = 1$ is half of the size of the SOM (the SOM is designed to have a 2-by-2 grid of output units), and $\sigma(t_{\max}) = 0.2$. Substituting these real values into eq. (3.33) gives

$$\sigma(t) = (0.2)^{\frac{t}{t_{\max}}} \tag{3.34}$$

Initialization of weights and $t_{\max}$: Functions that have been described so far may not be optimal, but they are usually sufficient for self-organizing features in SOMs [13]. We now answer the final question about how the weights are initialized at the beginning. It has been demonstrated that random initialization of weights may not be the *best* or *fastest* policy, the weighs finally coverage if $t_{\max}$ is big enough [12]. *A rule of thumb* for archiving statistical accuracy is that $t_{\max}$ should be at least 500 times the number of output units [12]. In this thesis, $t_{\max}$ is set up as 1000 times the number of output units.

Detailed Architecture and Algorithm

Linking every thing discussed in the previous subsection, we have the final design of the SOM as in fig. 3.15 on page 34 and the following algorithm for training it.

Algorithm 2: Training the SOM for Scoring Articles' Interestingness

---

Input : Untrained SOM,
Training data set $\mathcal{T} = \{(F(a), I(a))\}_{a=1}^{K}$ contains K articles a,
Weight change threshold W, maximum training passes $t_{\max}$,
Initial and final learning rates: $\alpha(t_0)$, $\alpha(t_{\max})$,
Half of the initial and final sizes of the neighboring area: $\sigma(t_0)$, $\sigma(t_{\max})$

Output: Trained SOM

1: *Initialization:*

- randomly choose values in the interval $[0.4, 0.6]$ for the initial weight vectors $\mathbf{w}_j(t_0)$.
- initialize the training pass $t = t_0 = 0$.

2: *Article Presenting and Best Matching Search:* for each article $a \in \mathcal{T}$, perform the following steps

- normalize input $F(a)$ into
$$i(a) = [i_a(k_1), i_a(k_2), \ldots, i_a(k_n), i_a(K_1), i_a(K_2), \ldots, i_a(K_m)]$$
and present it to the input layer.
- each input unit receives and forwards the corresponding value from $i(a)$ to all units in the hidden layer.
- find the winning unit c using eq. (3.27) in which $\mathbf{x}(t) = i(a)$.

3: *Weight Updating:*

- calculate neighboring units $N(c)(t)$ of c using $\sigma(t)$ and eq. (3.28).
- calculate the weight adjusting factor $h_{cj}(t)$ using eq. (3.30).
- for each units $j \in \{c\} \cup N(c)(t)$, adjust weight vectors associating with it using eq. (3.29) shown again as follows
$$\mathbf{w}_j(t) = \mathbf{w}_j(t) + h_{cj}(t)[\mathbf{x}(t) - \mathbf{w}_j(t)].$$

4: *Learning Rate and Neighboring Area's Size Tuning:*

- update learning rate $\alpha(t)$ using eq. (3.31) or its value-substituted eq. (3.32).
- update $\sigma(t)$ using eq. (3.33) or its value-substituted eq. (3.34).

5: *Continuation Condition Checking:* if the number of training passes $t > t_{\max}$ then stop the training; else increase $t = t + 1$, go back to step 2 *Article Presenting and Best Matching Search* and repeat for the next training pass.

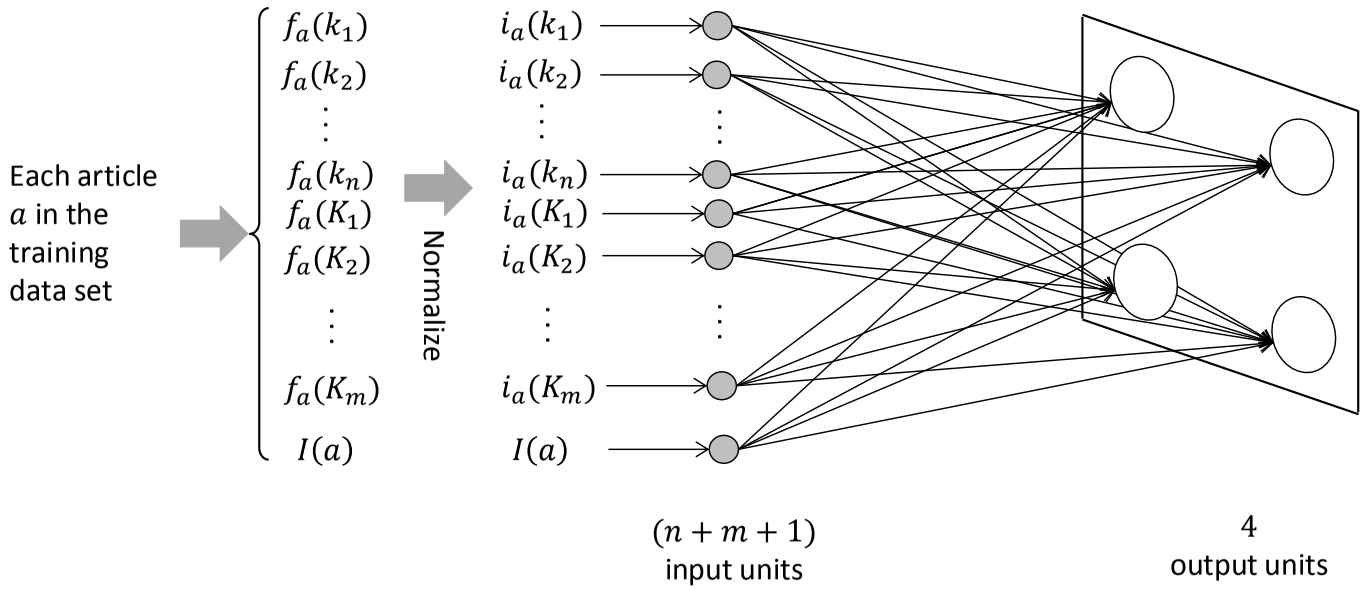


Figure 3.15: Detailed architecture of the SOM for scoring articles' interestingness. The output layer is a two-by-two lattice of four units that represent four clusters of articles whose interestingness drop in one of four intervals: $[0, 0.25)$, $[0.25, 0.5)$, $[0.5, 0.75)$ and $[0.75, 1.0]$. An extra input unit is added to receive $I(a)$ for each article a in the training data set $\mathcal{T}$ since $I(a)$s is known before the training step and we do not want to waste this information. After the SOM has been trained, a new article a_{new}'s interestingness is calculated by: first finding the winning unit as a_{new}'s input vector is presented to the trained SOM, and then the interestingness of a_{new} is assigned to the average of the interestingness of all articles that are classified as belong to the same winning unit of a_{new}.

3.4 Ranking Agents

We now consider how to calculate agents' reliability when they exchange information (articles and feedback) together. The calculation result is used to rank these agents and find out which ones are trustworthy and which ones are unreliable. If an agent receives feedback informing that its reliability is not high enough, it knows that the performance of sending articles to the lower level agent is not sufficient and may revise criterion in selecting articles from higher level agents so that the performance will be improved and hence the reliability will also be better.

3.4.1 Reliability of Each Agent

Figure 3.16 depicts two consecutive levels. A higher level agent X_i each time sends x_i articles to the lower level Y, Y will select only x_i^+ and eliminate x_i^- articles that are scored as highly interesting and not highly interesting respectively.

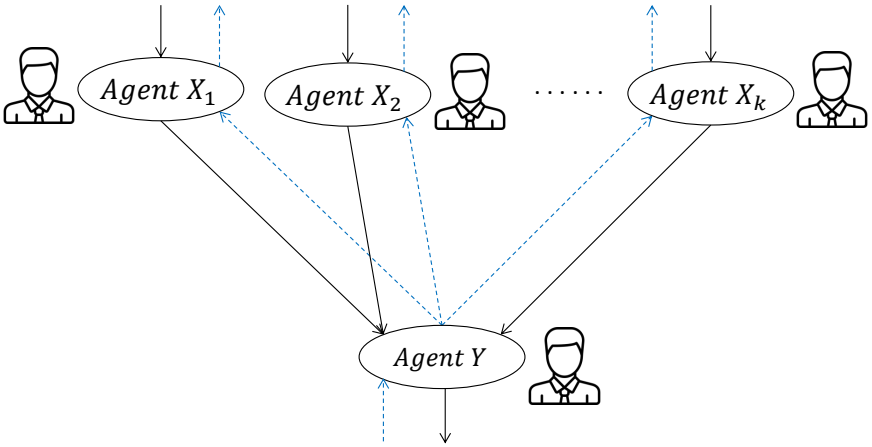


Figure 3.16: Agents in two consecutive levels exchange information. Higher level agents $X_1, X_2, \ldots, X_k$ send articles to and receive feedback from lower level agent Y. Each time agent X_i $(1 \leq i \leq k)$ sends a set of articles to agent Y, some articles in this set are selected as highly interesting while the others are eliminated by Y.

Let us drop the index i and consider an arbitrary agent X, at its n^{th} sending session, sends x_n articles to agent y. Let x_n^+ and x_n^- respectively denote the number of articles that are selected and the number of articles that are ignored (eliminated or unselected) by Y. A constraint between x_n^+ and x_n^- is $x_n = x_n^+ + x_n^-$. Performance, $r_n(X)$, at the n^{th} session of X can be evaluated by the following formula

$$r_n(X) = \frac{x_n^+}{x_n^+ + x_n^-} \tag{3.35}$$

However, eq. (3.35) only evaluates X's efficiency at the n^{th} session without considering the previous sessions. We therefore need to relate the performance $r_n(X)$ at session n and $r_k(X)$ at sessions k where $0 \leq k < n$ to compute the reliability of X during the entire exchanging process. Analyzing a series of sessions starting from the 0^{th}, the $1^{\text{st}}, \ldots,$ to the n^{th} in which reliability of each session is computed based on both the preceding sessions and the session itself we have:

Session 0:

$$r_0(X) = 0$$

Session 1:

$$r_1(X) = \frac{x_1^+}{x_1^+ + x_1^-}$$

.....

Session $n-1$:

$$r_{n-1}(X) = \frac{x_1^+ + x_2^+ + \ldots + x_{n-1}^+}{(x_1^+ + x_1^-) + (x_2^+ + x_2^-) + \ldots + (x_{n-1}^+ + x_{n-1}^-)}$$

Session n:

$$r_n(X) = \frac{x_1^+ + x_2^+ + \ldots + x_{n-1}^+ + x_n^+}{(x_1^+ + x_1^-) + (x_2^+ + x_2^-) + \ldots + (x_{n-1}^+ + x_{n-1}^-) + (x_n^+ + x_n^-)}$$

In order to rewrite $r_n(X)$ using $r_{n-1}(X)$ we compute the difference between their inversion as follows

$$\begin{aligned}
\frac{1}{r_n(X)} - \frac{1}{r_{n-1}(X)} &= \left[\frac{(x_1^+ + x_1^-) + (x_2^+ + x_2^-) + \ldots + (x_{n-1}^+ + x_{n-1}^-) + (x_n^+ + x_n^-)}{x_1^+ + x_2^+ + \ldots + x_{n-1}^+ + x_n^+}\right] - \\
&\quad \left[\frac{(x_1^+ + x_1^-) + (x_2^+ + x_2^-) + \ldots + (x_{n-1}^+ + x_{n-1}^-)}{x_1^+ + x_2^+ + \ldots + x_{n-1}^+}\right] \\
&= \left(1 + \frac{x_1^- + x_2^- + \ldots + x_{n-1}^- + x_n^-}{x_1^+ + x_2^+ + \ldots + x_{n-1}^+ + x_n^+}\right) - \left(1 + \frac{x_1^- + x_2^- + \ldots + x_{n-1}^-}{x_1^+ + x_2^+ + \ldots + x_{n-1}^+}\right) \\
&= \frac{X_{n-1}^- + x_n^-}{X_{n-1}^+ + x_n^+} - \frac{X_{n-1}^-}{X_{n-1}^+} \\
&\qquad \text{where } X_{n-1}^- = x_1^- + x_2^- + \ldots + x_{n-1}^- = \sum_{i=1}^{n-1} x_i^- \\
&\qquad \text{and} \quad X_{n-1}^+ = x_1^+ + x_2^+ + \ldots + x_{n-1}^+ = \sum_{i=1}^{n-1} x_i^+ \\
&= \frac{X_{n-1}^- X_{n-1}^+ + x_n^- X_{n-1}^+ - X_{n-1}^- X_{n-1}^+ - X_{n-1}^- x_n^+}{\left(X_{n-1}^+ + x_n^+\right) X_{n-1}^+} \\
&= \frac{x_n^- X_{n-1}^+ - X_{n-1}^- x_n^+}{\left(X_{n-1}^+ + x_n^+\right) X_{n-1}^+}.
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \frac{1}{r_n(X)} &= \frac{1}{r_{n-1}(X)} + \frac{x_n^- X_{n-1}^+ - X_{n-1}^- x_n^+}{\left(X_{n-1}^+ + x_n^+\right) X_{n-1}^+} \\
&= \frac{1}{r_{n-1}(X)} + \frac{x_n^- \dfrac{X_{n-1}^+}{X_{n-1}^+ + X_{n-1}^-} - \left(\dfrac{X_{n-1}^+ + X_{n-1}^-}{X_{n-1}^+ + X_{n-1}^-} - \dfrac{X_{n-1}^+}{X_{n-1}^+ + X_{n-1}^-}\right) x_n^+}{\left(X_{n-1}^+ + x_n^+\right) \dfrac{X_{n-1}^+}{X_{n-1}^+ + X_{n-1}^-}} \\
&= \frac{1}{r_{n-1}(X)} + \frac{x_n^- r_{n-1}(X) - [1 - r_{n-1}(X)]\, x_n^+}{\left(X_{n-1}^+ + x_n^+\right) r_{n-1}(X)}
\end{aligned}$$

$$\Rightarrow \frac{1}{r_n(X)} = \frac{\left(X_{n-1}^{+} + x_n^{+}\right) + x_n^{-} r_{n-1}(X) - \left[1 - r_{n-1}(X)\right] x_n^{+}}{\left(X_{n-1}^{+} + x_n^{+}\right) r_{n-1}(X)}$$

$$= \frac{X_{n-1}^{+} + (x_n^{+} + x_n^{-}) r_{n-1}(X)}{\left(X_{n-1}^{+} + x_n^{+}\right) r_{n-1}(X)}.$$

Therefore, we have the following equation that shows the relation between $r_n(X)$ and $r_{n-1}(X)$

$$r_n(X) = \frac{\left(X_{n-1}^{+} + x_n^{+}\right) r_{n-1}(X)}{X_{n-1}^{+} + (x_n^{+} + x_n^{-}) r_{n-1}(X)} \tag{3.36}$$

where:

$(X_{n-1}^{+} + x_n^{+})$	is the total number of articles that are cumulatively selected by Y from session 0 to session n,
$r_{n-1}(X)$	is the reliability of X evaluated by Y after session $n-1$ has been finished,
X_{n-1}^{+}	is the total number of articles that are cumulatively selected by Y from session 0 to session $n-1$,
$(x_n^{+} + x_n^{-})$	is the total number of articles sent by X during session n.

3.4.2 Reliability Updating Tables

Equation (3.36) suggests us that if a *receiver agent* Y wants to compute reliability $r_n(X)$ of a *sender agent* X after session n is conducted, all Y has to memorize are only X_{n-1}^{+} and $r_{n-1}(X)$. The reason is that at session n, after receiving x_n articles from X, Y will check and select x_n^{+} articles and ignore x_n^{-} remaining ones. Thus, at session n, Y knows what exactly x_n^{+} and x_n^{-} are by itself. Y therefore can compute all terms in eq. (3.36) that is used to measure X's reliability at an arbitrary session n. These analyses lead us to the design of the following *reliability updating table* that is created, kept and updated by agent Y to measure reliability of agents $X_1, X_2, \ldots, X_k$ as modeled in fig. 3.16.

Agent	Session	Reliability	Total num. of selected articles
X_1	$n_1 - 1$	$r_{n_1-1}(X_1)$	$(X_1)^{+}_{n_1-1}$
X_2	$n_2 - 1$	$r_{n_2-1}(X_2)$	$(X_2)^{+}_{n_2-1}$
...	...	...	...
X_k	$n_k - 1$	$r_{n_k-1}(X_k)$	$(X_k)^{+}_{n_k-1}$

Update

Agent	Session	Reliability	Total num. of selected articles
X_1	n_1	$r_{n_1}(X_1)$	$(X_1)^{+}_{n_1}$
X_2	n_2	$r_{n_2}(X_2)$	$(X_2)^{+}_{n_2}$
...	...	...	...
X_k	n_k	$r_{n_k}(X_k)$	$(X_k)^{+}_{n_k}$

Figure 3.17: Reliability updating table. Design of a table that agent Y uses to calculate reliability of sender agents $X_1, X_2, \ldots, X_k$ as depicted in fig. 3.16. At step $n_i - 1$, Y saves $r_{n_i-1}(X_i)$ and $(X_i)^{+}_{n_i-1}$ for agents X_i where $1 \leq i \leq k$ is an index over the set of k sender agents. When X_i send articles to Y in the next session n_i, Y will update their reliability $r_{n_i-1}(X)$ to $r_{n_i}(X)$ using eq. (3.36) and their total numbers of articles, $(X_i)^{+}_{n_i} = (X_i)^{+}_{n_i-1} + x^{+}_{n_i}$, that are selected as highly interesting from the beginning to session n_i. Note that n_i where $1 \leq i \leq k$ are not necessary to be equal since some agents may be active and do many sending sessions while the others may be not.

Agent Y with the reliability updating table described in fig. 3.17 can rank agents $X_1, X_2, \ldots, X_k$ based on the third column Reliability, e.g. sorting table rows by comparing

values in the third column. Using the design that have been discussed so far, we now turn back to our example introduced in section 3.2 and modify the reliability updating tables in fig. 3.2 as well as fig. 3.3. The modification result is shown in the following fig. 3.18.

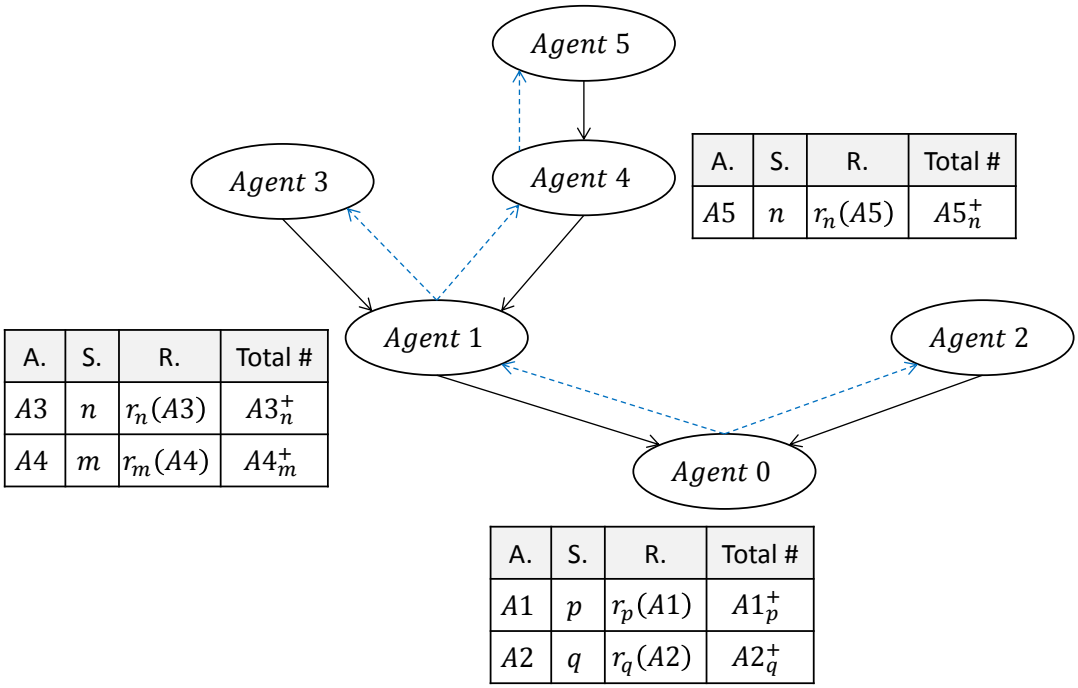


Figure 3.18: An example of the general system model with modified version of reliability updating tables. $A1, A2, \ldots, A5$ are the names of 6 agents. Three reliability updating tables are created and owned by *Agent* 0, *Agent* 1 and *Agent* 4. A., S., R. and Total # are abbreviations of the reliability updating table's column headings in fig. 3.17. With these tables, *Agent* 0 is able to find out in *Agent* 1 and *Agent* 2 which one is more reliable, *Agent* 1 is able to find out in *Agent* 3 and *Agent* 4 which one is more reliable, and *Agent* 4 is able to judge *Agent* 5's reliability numerically.

3.4.3 Trustworthiness of Filtered Information

We now turn to our final question: *how trustworthy is the information that is filtered by the agent network?* In other words, how much can we trust in the filtering result returned by the root agent? In order to answer the question, this thesis proposes a naive method in which each non-leaf-node agent's reliability updating table is used to naively convert into a conditional probability table as in the following fig. 3.19

Agent	Session	Reliability	Total num. of selected articles
X_1	n_1	$r_{n_1}(X_1)$	$(X_1)^+_{n_1}$
X_2	n_2	$r_{n_2}(X_2)$	$(X_2)^+_{n_2}$
...	...	...	...
X_k	n_k	$r_{n_k}(X_k)$	$(X_k)^+_{n_k}$

naively convert

X_1	X_2	...	X_k	$P(Y\|X_1, X_2, \ldots, X_k)$
r	r	...	r	$P(Y = r\|X_1 = r, X_2 = r, \ldots, X_k = r)$
r	r	...	u	$P(Y = r\|X_1 = r, X_2 = r, \ldots, X_k = u)$
...	...	...	...	.....
u	u	...	r	$P(Y = r\|X_1 = u, X_2 = u, \ldots, X_k = r)$
u	u	...	u	$P(Y = r\|X_1 = u, X_2 = u, \ldots, X_k = u)$

2^k rows

r: reliable u: unreliable

Figure 3.19: Naive conversion of reliability updating table into conditional probability table.

The conditional probability table on the right hand side in fig. 3.19 consists of k columns for k sender agents and a column for the conditional probability of Y is reliable given the sender agents' reliability states (r and u stand for two possible reliability states: *reliable* and *unreliable* respectively). Each agent X_i's reliability state is decided based on it reliability $r_{n_i}(X_i)$. If $r_{n_i}(X_i) \geq \theta$ where $\theta \in [0, 1]$ is a threshold showing the minimum value of reliability an agent should be higher to become reliable, then X_i is considered as reliable; otherwise it is unreliable.

The probability of an agent X_i is reliable can be calculated by counting the number of sessions in which its reliability at these sessions is greater than or equal to θ

$$P(X_i = r) = \frac{|\{s \mid 1 \leq s \leq n_i, r_s(X_i) \geq \theta\}|}{n_i} \tag{3.37}$$

where $|.|$ is the set size operator, n_i is the most recent session that have been done, s is an index over sessions $1, 2, \ldots, n_i$. Obviously, the following equation holds

$$P(X_i = u) = 1 - P(X_i = r). \tag{3.38}$$

Let us first reconsider the example of the general model that added the reliability updating tables as in fig. 3.18. Three conditional probability tables that are converted from these reliability updating tables are added to three corresponding agents. In addition, a probability table is added for each leaf-node agent with the purpose of saving its probability of being reliable. The resulting model is illustrated in the following figure

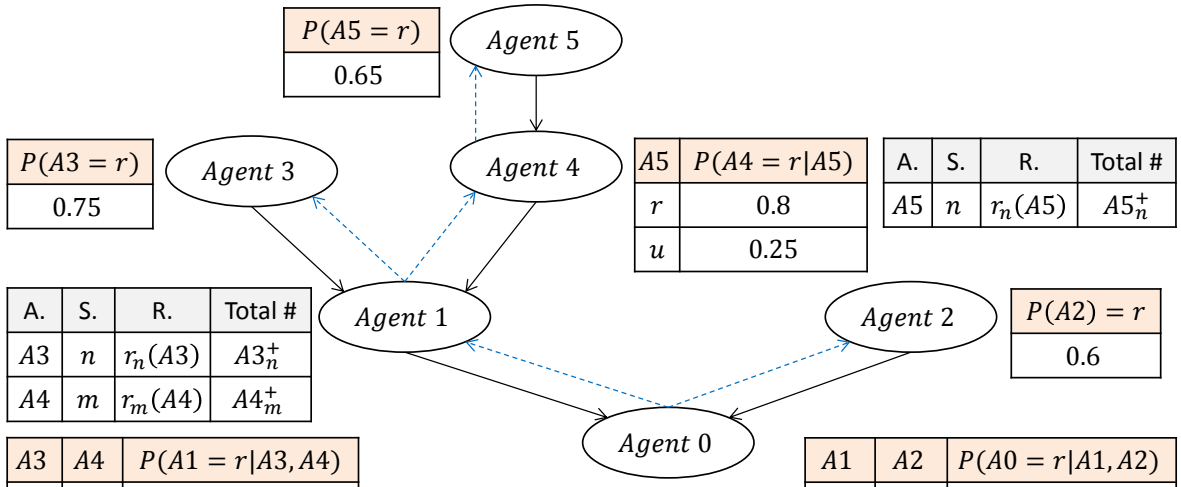


Figure 3.20: An example of the general system model with conditional probability tables. Three conditional probability tables for *Agent* 0, *Agent* 1 and *Agent* 4, and three probability tables for *Agent* 3, *Agent* 5, *Agent* 2 are added to the example model that is previously shown in fig. 3.18. $A0, A1, \ldots, A5$ are the abbreviated names of agents and are used as variables in probability formulas. Numeric values are given to make both conditional probability tables and probability tables concrete, i.e. $P(A0 = r|A1 = r, A2 = r) = 0.95$ is the probability of $A0$ is reliable (information provided by $A0$ is trustworthy) given both $A1$ and $A2$ are reliable. $P(A0 = u|A1 = r, A2 = r) = 1 - P(A0 = r|A1 = r, A2 = r) = 0.05$.

Answering the question *how trustworthy is the information (articles) filtered by the agent network* in fig. 3.20 is equivalent to answering what is the probability of all agents in the network are reliable. The calculation of $P(A0, A1, A2, A3, A4, A5)$ where each each variable may be r or u is the more general approach for answer a wider range of questions regarding to the accuracy of the network.

In order to compute $P(A0, A1, A2, A3, A4, A5)$, we assume that each agent's reliability depends only on the higher-adjacent-level agents that directly connect to it (the connections are depicted as black arrows in fig. 3.20), and is independent from all others, i.e. *Agent* 1's reliability depends only on *Agent* 3 and *Agent* 4. This assumption is reasonable because

- all articles received from *Agent* 5 are rechecked, scored, and selected by *Agent* 4, so the quality of articles that *Agent* 4 sends to *Agent* 1 is centrally decided by *Agent* 4. It is therefore suitable for assuming that *Agent* 1's reliability depends on *Agent*4. Furthermore, *Agent* 1's reliability also depends on *Agent* 3 since it plays the same role of sending articles as *Agent* 4.
- obviously, *Agent* 1's reliability is independent of both *Agent* 0 and *Agent* 2 since no article is sent from these two agents to *Agent* 1.

With the above assumption, we have

$$\begin{aligned}
&P(A0|A1, A2, A3, A4, A5) = P(A0|A1, A2),\\
&P(A1|A2, A3, A4, A5) = P(A1|A3, A4),\\
&P(A2|A3, A4, A5) = P(A2)\\
&P(A3|A4, A5) = P(A3)
\end{aligned}$$

The chain rule[12] in probability theory gives

$$\begin{aligned}
P(A0, A1, A2, A3, A4, A5) = {} & P(A0|A1, A2, A3, A4, A5) \times P(A1|A2, A3, A4, A5)\times\\
& P(A2|A3, A4, A5) \times P(A3|A4, A5) \times P(A4|A5) \times P(A5)\\
= {} & P(A0|A1, A2) \times P(A1|A3, A4)\times\\
& P(A2) \times P(A3) \times P(A4|A5) \times P(A5). \qquad (3.39)
\end{aligned}$$

Using eq. (3.39) and referring to the conditional probability tables in fig. 3.20, we can compute P(A0, A1, A2, A3, A4, A5) for all $2^6 = 64$ value assignments of 6 variables $A0, A1, A2, A3, A4, A5$. For example,

$$\begin{aligned}
&P(A0 = r, A1 = r, A2 = r, A3 = r, A4 = r, A5 = r)\\
&\quad = P(A0 = r|A1 = r, A2 = r) \times P(A1 = r|A3 = r, A4 = r)\times\\
&\quad\quad P(A2 = r) \times P(A3 = r) \times P(A4 = r|A5 = r) \times P(A5 = r)\\
&\quad = 0.95 \times 0.9 \times 0.6 \times 0.75 \times 0.8 \times 0.65 = 0.20007
\end{aligned}$$

is the probability of the network is trustworthy with every agent is reliable.

[12] the chain rule expresses probability of the conjunction of a set of variables using conditional probabilities

$$P(x_1, x_2, \ldots, x_n) = P(x_n|x_{n-1}, \ldots, x_1)P(x_{n-1}|x_{n-2}, \ldots, x_1)\cdots P(x_2|x_1)P(x_1) = \prod_{i=1}^{n} P(x_i|x_{i-1}, \ldots, x_1)$$

where $P(x_1, x_2, \ldots, x_n)$ is the probability of x_1 and x_2 and ... and x_n; $P(x_n|x_{n-1}, \ldots, x_1)$ is the conditional probability of x_n given that x_{n-1} and ... and x_1.

$$
\begin{aligned}
&P(A0 = r, A1 = r, A2 = u, A3 = u, A4 = r, A5 = r) \\
&\quad = P(A0 = r|A1 = r, A2 = u) \times P(A1 = r|A3 = u, A4 = r) \times \\
&\quad\quad P(A2 = u) \times P(A3 = u) \times P(A4 = r|A5 = r) \times P(A5 = r) \\
&\quad = 0.7 \times 0.55 \times (1 - 0.6) \times (1 - 0.75) \times 0.8 \times 0.65 = 0.09009
\end{aligned}
$$

is the probability of the network is trustworthy with all agents are reliable except two leaf node agents, *Agent* 2 and *Agent* 3, are unreliable. It is easy to observe that the network's trustworthiness in this case is relatively low compared with the case all agents are reliable.

Chapter 4

Experiments

In this chapter, results obtained from experimental evaluation of our proposed system are reported. We describe the datasets used, experimental methodology and performance measures that are appropriate for verifying experiment evaluation.

4.1 Generate Datasets for Trainings Agents

We created 8 training datasets representing 8 research areas that 8 intelligent agents will learn to be able to score the interestingness of articles in these areas. Table A.1 shows lists of raw keywords selected to describe the 8 research areas. Each compound raw keyword is then split into smaller single words to be counted appearing frequency in each article. We will call the single words as *keywords* from now on. The list of research areas, their *keywords* and numbers of keywords are summarized in table A.2.

Table 4.1: Training datasets' sizes.

#	Research Area	Training Dataset's Size
1	Artificial Intelligence	1012
2	Neural Networks	1163
3	Deep Learning	1043
4	Computer Networks	1247
5	Natural Language Processing	1375
6	Logic	849
7	Fuzzy Logic	886
8	Modal Logic	911

Since judging interestingness is subjective and depends on individual's interest, in order to keep the objectivity of our experiments, we randomly generated 8 training data sets with the number of data vectors (input vectors of agents' neural networks) in each dataset is reported in table 4.1. The criteria for creating each data vector are: (*i*) the more frequently the keywords appear, the higher the interestingness would be but not all the cases, (*ii*) only a reasonable number of keywords should be allowed because an article containing all keywords in the list is not practical. Figure 4.1 on page 43 graphically visualizes distributions of data vectors in 8 training datasets. We can see that the distributions satisfy the two required criteria mentioned. Our agents, therefore, if are trained

with these training datasets, will also have the ability of reflecting the interestingness in relation with the number of appearing keywords.

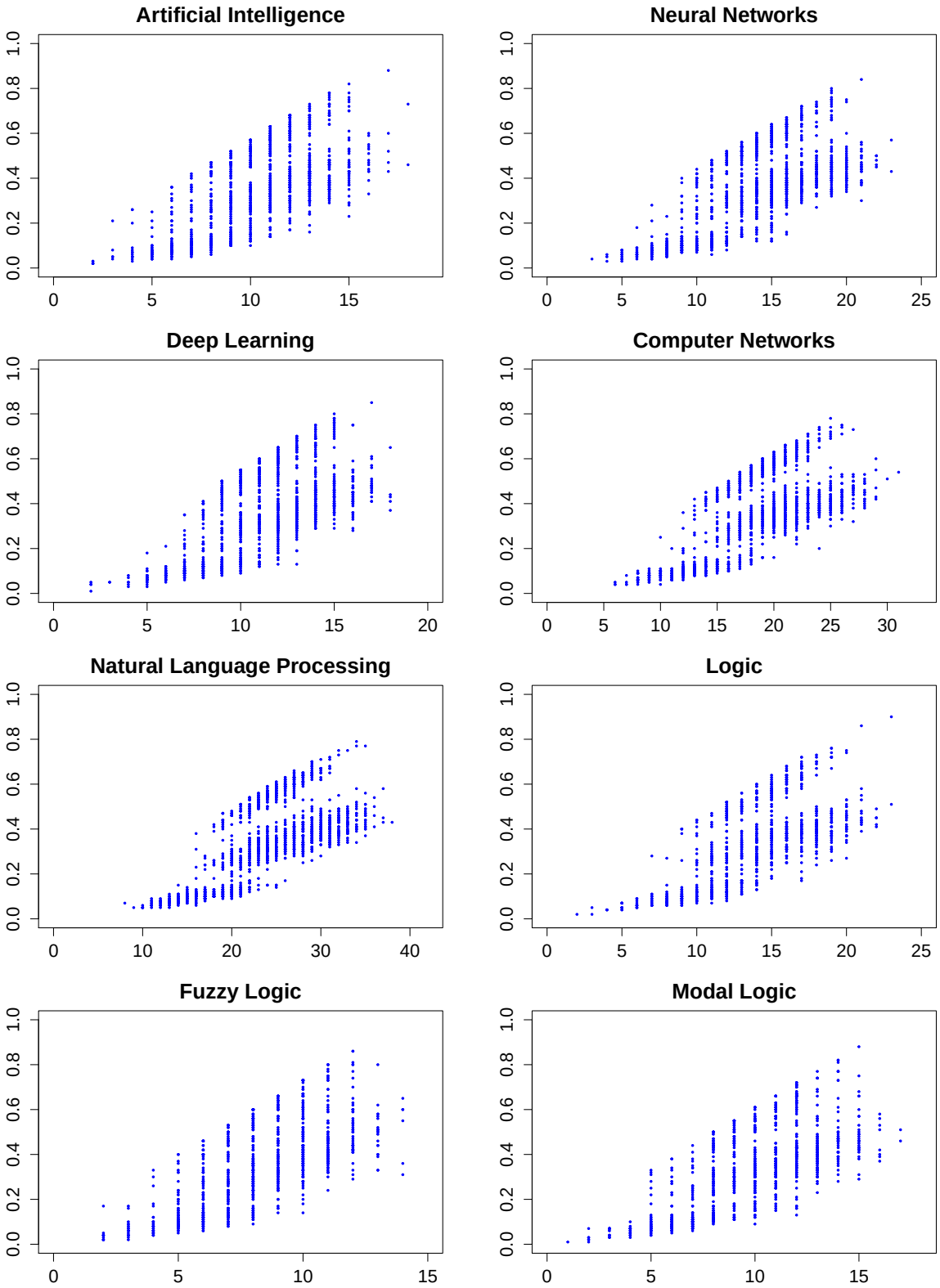


Figure 4.1: Distributions of input vectors in training datasets. The vertical and horizontal axes represent the interestingness and the number of appearing keywords, respectively.

After randomly generating a vector of keyword frequencies $[f(k_1), f(k_2), \ldots, f(k_n)]$ for each article that is assumed to exist in practice, we calculate the interestingness I of the

article as follows

$$I = \frac{1}{n}\sum_{x=1}^{n} i(k_x)$$

where $i(k_x)$ is computed using eq. (3.20) in which $0 \leq f(k_x) \leq 160$ and $\theta(k_x) = 10$ for all $x = 1, 2, \ldots, n$.

The reason why we chose data vectors be randomized is not only to keep our experiments objective but it is also because we have already split the list of raw keywords into single words without paying attention to the natural language relations between them. However, in vector space models, even though the use of natural language understanding for measuring text's feature is not considered, the use of words has been found to be quite effective [11].

4.2 Conduct Experiments with Single Agents

In order to verify that agents were trained to have sufficient knowledge and are able to score the interestingness of articles that they have never been presented before, we ran 8 experiments and obtained the following result:

Table 4.2: Experiments with single agents. Δ is the abstract difference between the interestingness returned by agents and the interestingness assigned to each article before conducting experiments.

Agent majoring in	**Number of Articles Scored**	**Accuracy (%) in the case**			
		$\Delta < 0.1$	$\Delta < 0.15$	$\Delta < 0.2$	$\Delta < 0.25$
Artificial Intelligence	47	36.17	53.19	63.83	76.60
Neural Networks	71	36.62	54.93	69.01	76.06
Deep Learning	44	36.36	61.36	70.45	72.73
Computer Networks	48	29.17	52.08	58.33	64.58
Natural Language Processing	60	31.67	55.00	66.67	70.00
Logic	62	25.81	54.84	67.74	69.35
Fuzzy Logic	54	25.93	48.15	66.67	77.78
Modal Logic	55	27.27	43.64	63.64	72.72
Average	55	31.13	52.90	65.79	72.48

In order to evaluate the above result, the *probabilistic relevance framework BM25* was chosen to compare with our neural network framework. BM25 or sometimes called Okapi BM25 is a formal framework for document retrieval that had ever been the most successful text-retrieval algorithms [23]. The BM25 function $score(a, K)$ that is used to score the relevant (we consider as the interestingness) of each article a with respect to the query $K = [k_1, k_2, \ldots, k_n]$ (the list of keywords) is as follows

$$score(a, K) = \sum_{i=1}^{n} \text{IDF}(k_i) \frac{f_a(k_i)(c+1)}{f_a(k_i) + c\left(1 - b + b\dfrac{dl}{avgdl}\right)}, \tag{4.1}$$

where $f_a(k_i)$ is the frequency of keyword k_i in the article a, dl is the the length of a in words, $avgdl$ is the average length in words of all articles in the testing dataset. c and b are two free parameters. Normally, $0 < b < 0.8$ and $1.2 < \text{c} < 2$. $\text{IDF}(k_i)$ is the *inverse*

document frequency weight of the keyword k_i and is computed as follows

$$\text{IDF}(k_i) = \log \frac{N - n(k_i) + 0.5}{n(k_i) + 0.5}, \tag{4.2}$$

where N is the total number of articles in the testing dataset, and $n(k_i)$ is the number of articles containing k_i.

8 experiments in 8 research areas were conducted using the BM25 function with 8 testing databases same as in experiments with agents. In each experiment, b is fixed with value of 0.75 while c is assigned with three values: 1.4, 1.6 and 1.8. The average of three values returned by function $score(a, K)$ for three values of c is computed and normalized into a value in the interval $[0, 1]$. Result of using the BM25 function is summarized in the following table

Table 4.3: Experiments with Okapi BM25 function. Δ is the abstract difference between the interestingness returned by the Okapi BM25 function and the interestingness assigned to each article before conducting experiments.

Agent majoring in	**Number of Articles Scored**	**Accuracy (%) in the case**			
		$\Delta < 0.1$	$\Delta < 0.15$	$\Delta < 0.2$	$\Delta < 0.25$
Artificial Intelligence	47	29.79	44.68	59.57	68.09
Neural Networks	71	12.68	21.13	30.99	36.62
Deep Learning	44	11.36	25.00	29.55	47.73
Computer Networks	48	10.42	20.83	25.00	27.08
Natural Language Processing	60	23.33	30.00	35.00	43.33
Logic	62	8.06	12.90	16.13	25.81
Fuzzy Logic	54	42.59	59.26	68.52	81.48
Modal Logic	55	12.73	21.82	38.18	61.82
Average	55	18.87	29.45	37.87	49.00

The results obtained from experiments with single agents in table 4.2 and with the Okapi BM25 ranking function in table 4.3 provide us strong evidence that by using neural networks, agents score articles' interestingness with the accuracy higher than the traditional popular BM25 framework. In 8 experiments, there are 7 in which our agents' performance is much better than function *score*. Furthermore, the average accuracy of our agents in the case *error* $\Delta < 0.2$ that is acceptable in practice is approximately 1.7373 times higher than the average accuracy of the BM25 function. This suggests us that randomly generating training datasets so that agents can learn users' preferences meets the requirement of saving time while still ensuring acceptable accuracy.

Chapter 5

Conclusions and Future Directions

5.1 Conclusions

In this research, we have proposed a model of multi-agent systems using neural networks to evaluate the new coming information and using probabilistic reasoning in combination with reliability updating mechanism to manage agents' performance. The flexibility of the system was demonstrated via showing that both supervised and unsupervised learnings can be integrated into the model without affecting the process of agent ranking. The reliability updating tables were proposed with the formula on how to calculate and update agents' reliability. Introducing these tables also provides us a mathematical and systematical way to determine the trustworthiness of filtered information. The final contribution of this research is that we have shown by using neural network to score the interestingness of research articles, the accuracy obtained is approximately 1.7 times higher than using the popular BM25 function and is about 31.13% to 65.79%.

5.2 Future Work

The next directions of our research inheriting from this research include (i) improve the accuracy of scoring interestingness by considering the natural language relations between keywords - the things that have been ignored, (ii) apply the designed model to other areas and deploy the developed system to more users, (iii) continue the research on applying probabilistic dynamic epistemic logic and hybrid logic in representing the entire information filtering process in logical formulas.

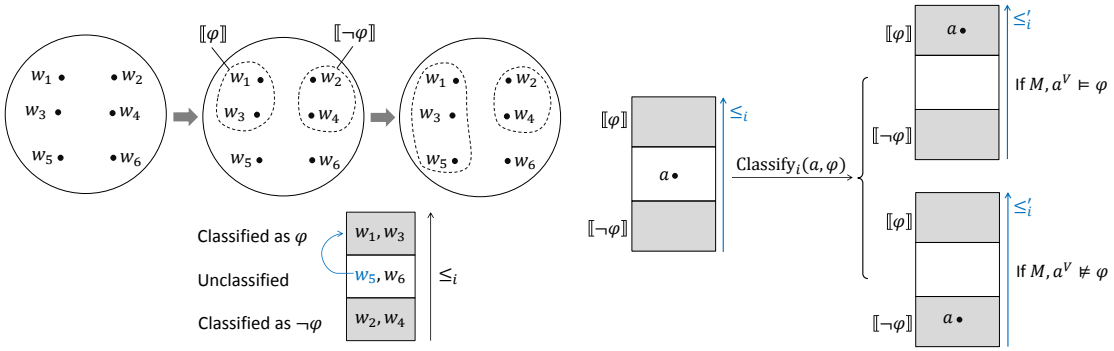


Figure 5.1: Future research direction on representing the entire multi-agent system of information filtering by logical formulas. The idea is to represent the system using logical model in which agents' reliability is formalized by plausibility relation. When an object's label is determined, the model is updated.

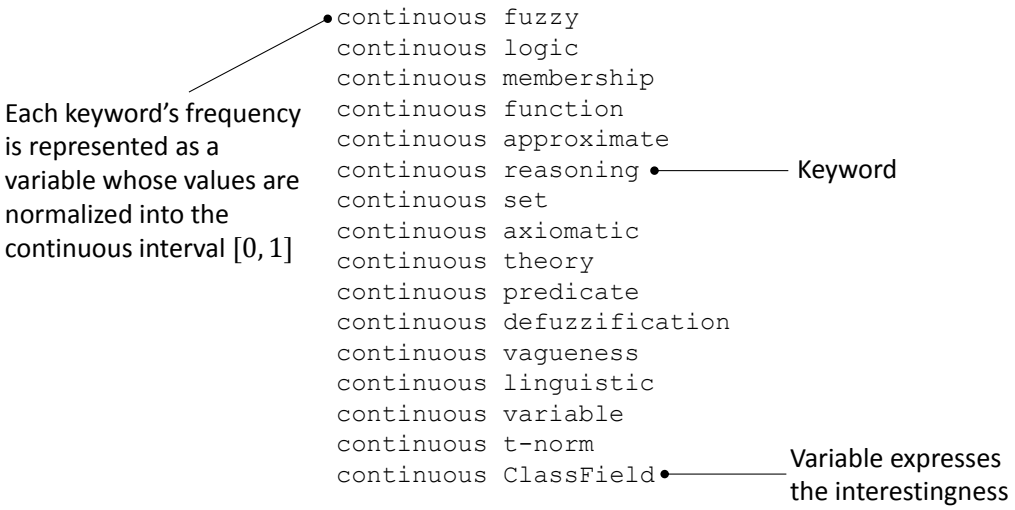
Appendix A

Training Datasets

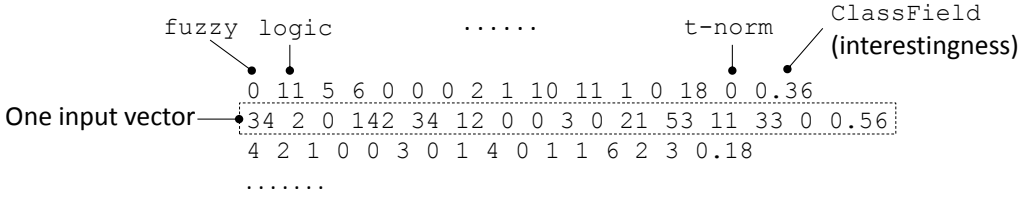
This appendix provides the details of training datasets used in all experiments of this research. All training datasets are available in directory `TrainingDataSets` at the following url

https://github.com/TranXuanHoang/InformationFilteringSystem

Each training dataset consists of a data definition file `infofilter.dfn` defining the list of keywords that describe the user's research area, a data file `infofilter.dat` containing input vectors for training neural networks and a returned-after-training file `filterAgent.ser`. For example, in the dataset for training an agent majoring in *fuzzy logic*, the `infofilter.dfn` file that defines variables expressing the list of keywords (see table A.2) and the interestingness is as follows



File `infofilter.dat` saves input vectors with each in one line as the following description



File `filterAgent.ser` contains all information about the backpropagation neural network and the self-organizing map that have been trained. In order to avoid having to train the neural networks again, just place this file in the same directory where the information filtering application (see appendix B) is run in the next time.

Table A.1: List of research areas and corresponding raw keywords used to generate training datasets. **(RK) Number of Raw Keywords**. 8 training datasets are generated to train 8 agents which major in 8 different research areas. Raw keywords are given to describe these areas. The terminology *raw keywords* refers to originally given keywords (phrases) that are not split into smaller words.

#	Research Area	(RK)	List of Raw Keywords
1	Artificial Intelligence	14	artificial intelligence, machine learning, deep learning, reinforcement learning, neural network, intelligent agent, knowledge, reasoning, learning, planning, decision, natural language processing, classification, robotic
2	Neural Networks	18	neural network, multilayer perceptron, backpropagation, self-organizing map, convolutional neural network, recurrent neural network, weight update, gradient descent, adaptive learning, competitive learning, supervised learning, unsupervised learning, online learning, batch learning, hidden layer, unit, activation, feed-forward
3	Deep Learning	11	deep learning, multiple processing, multilayer neural network, convolutional neural network, backpropagation algorithm, supervised learning, reinforcement learning, recurrent neural networks, stochastic gradient descent, deep feedforward, pattern recognition
4	Computer Networks	25	computer networks, Internet, application layer, presentation layer, session layer, transport layer, network layer, data link layer, physical layer, router, gateway, bridge, port, socket, IP address, MAC address, TCP/IP, UDP, protocol, wireless LAN, DNS, link state routing algorithm, distance vector routing algorithm, multimedia, security
5	Natural Language Processing	24	natural language processing, text classification, word, lexical, dictionary, bilingual corpus, query, regular expression, n-gram tagging, syntax, context free grammar, feature structure, information extraction, information retrieval, part-of-speech tagging, hidden Markov models, precision, recall, question answering, automata, parse tree, Chomsky normal form, CKY algorithm, machine translation
6	Logic	22	logic, validity, completeness, soundness, truth table, true, false, formal deduction, propositional logic, first-order logic, negation, conjunction, disjunction, implication, all quantifier, some quantifier, syntax, semantics, premise, conclusion, tautology, theorem
7	Fuzzy Logic	10	fuzzy logic, membership function, approximate reasoning, fuzzy set, axiomatic fuzzy set theory, fuzzy predicate logic, defuzzification, vagueness, linguistic variable, t-norm
8	Modal Logic	12	modal logic, knowledge operator, belief operator, Kripke model, posibility, necessary, modal axiom, proof system, sound, complete, possible world semantic, consistent

Table A.2: Keyword lists for generating training datasets. **(K) Number of Keywords**. Each list of keywords is created from its corresponding list of raw keywords in table A.1 by splitting compound raw keywords into single words and keeping only one word if the word is duplicated.

#	Research Area	(K)	List of Keywords
1	Artificial Intelligence	19	artificial, intelligence, machine, learning, deep, reinforcement, neural, network, intelligent, agent, knowledge, reasoning, planning, decision, natural, language, processing, classification, robotic
2	Neural Networks	25	neural, network, multilayer, perceptron, backpropagation, self-organizing, map, convolutional, recurrent, weight, update, gradient, descent, adaptive, learning, competitive, supervised, unsupervised, online, batch, hidden, layer, unit, activation, feed-forward
3	Deep Learning	20	deep, learning, multiple, processing, multilayer, neural, network, convolutional, backpropagation, algorithm, supervised, reinforcement, recurrent, networks, stochastic, gradient, descent, feedforward, pattern, recognition
4	Computer Networks	33	computer, networks, Internet, application, layer, presentation, session, transport, network, data, link, physical, router, gateway, bridge, port, socket, IP, address, MAC, TCP/IP, UDP, protocol, wireless, LAN, DNS, state, routing, algorithm, distance, vector, multimedia, security
5	Natural Language Processing	42	natural, language, processing, text, classification, word, lexical, dictionary, bilingual, corpus, query, regular, expression, n-gram, tagging, syntax, context, free, grammar, feature, structure, information, extraction, retrieval, part-of-speech, hidden, Markov, models, precision, recall, question, answering, automata, parse, tree, Chomsky, normal, form, CKY, algorithm, machine, translation
6	Logic	25	logic, validity, completeness, soundness, truth, table, true, false, formal, deduction, propositional, first-order, negation, conjunction, disjunction, implication, all, quantifier, some, syntax, semantics, premise, conclusion, tautology, theorem
7	Fuzzy Logic	15	fuzzy, logic, membership, function, approximate, reasoning, set, axiomatic, theory, predicate, defuzzification, vagueness, linguistic, variable, t-norm
8	Modal Logic	18	modal, logic, knowledge, operator, belief, Kripke, model, posibility, necessary, axiom, proof, system, sound, complete, possible, world, semantic, consistent

Appendix B

The Information Filtering Application

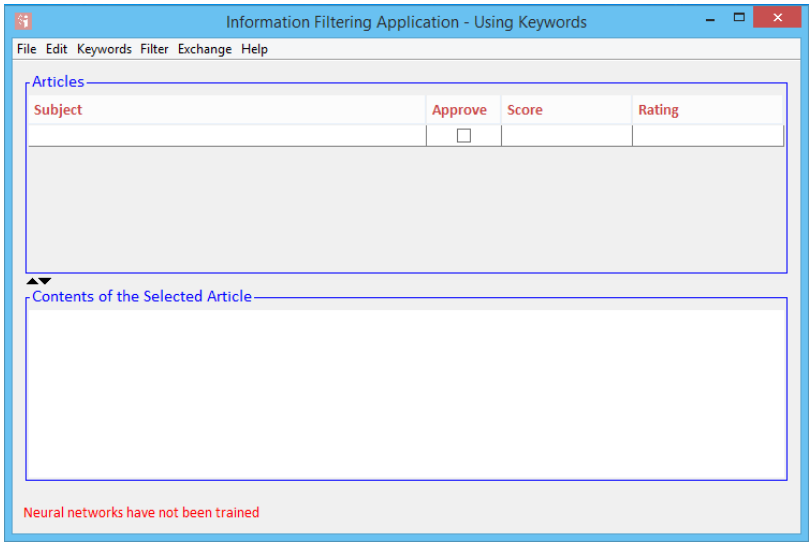
This appendix introduces an information filtering application developed by this thesis's author as a demonstration for all concepts, designs and algorithms we have discussed in the previous chapters. Key functions of the application will mainly be explained with the purpose of providing manual instructions on how to use the application.

The application was developed in Java. Source code and runnable `.jar` app can be downloaded from the following url:

https://github.com/TranXuanHoang/InformationFilteringSystem

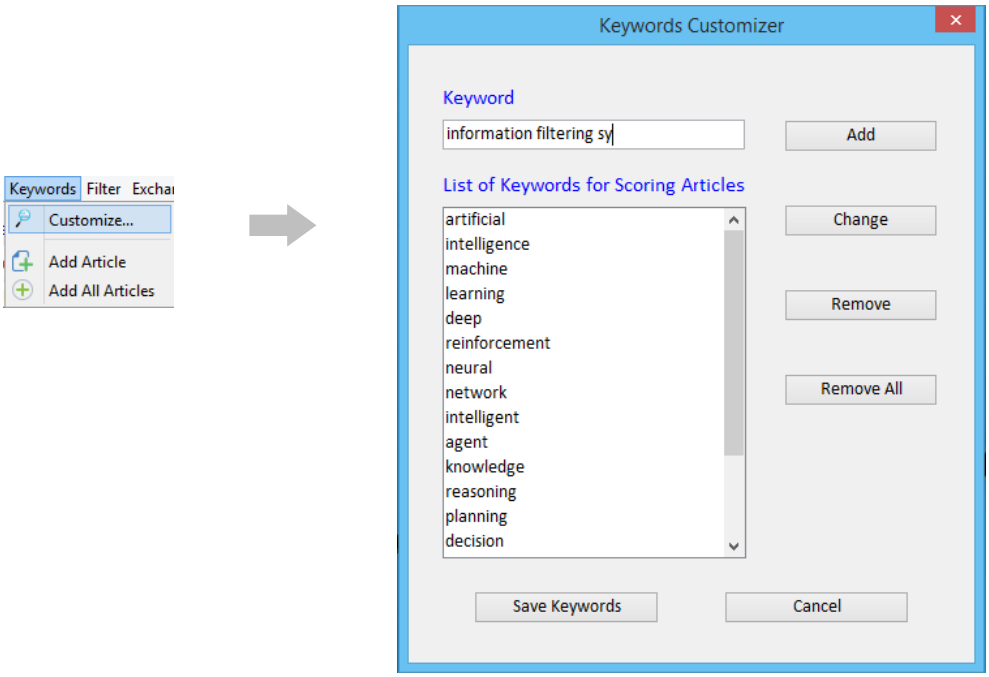
Source code:	in directory `src` from the above url
Runnable file:	in directory `app` from the above url

Starting Window: when the application is started, its main graphical user interface (GUI) is as the following figure. The top half of the GUI is a table containing information

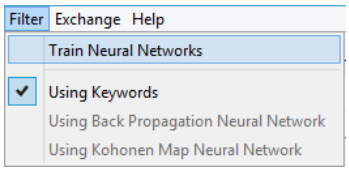


of articles that will be loaded from personal computer or downloaded from the Internet. `Subject` column shows articles' subjects, `Approve` column contains check boxes so that user can tick on to indicate which articles are selected to be sent to other user. `Score` column is the interestingness of articles and is automatically computed by the app. `Rating` column allows user to score articles' interestingness again if the app's returned `scores` is not correct.

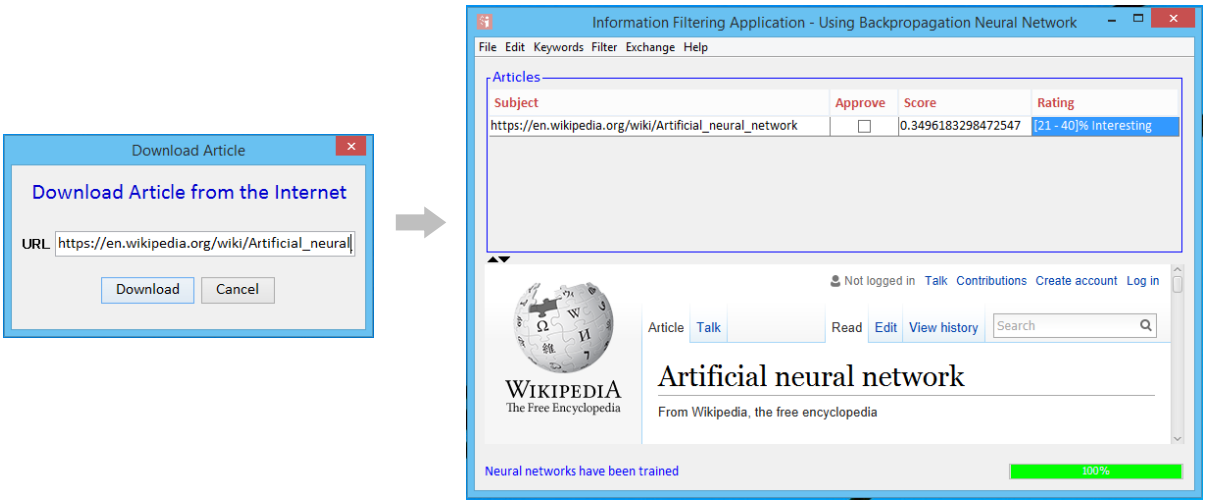
Add, Change, Remove Keywords: select `Keywords > Customize...`, then a dialog box will appear and keywords can be added, changed or removed. Hit `Save Keywords` button to save any change.



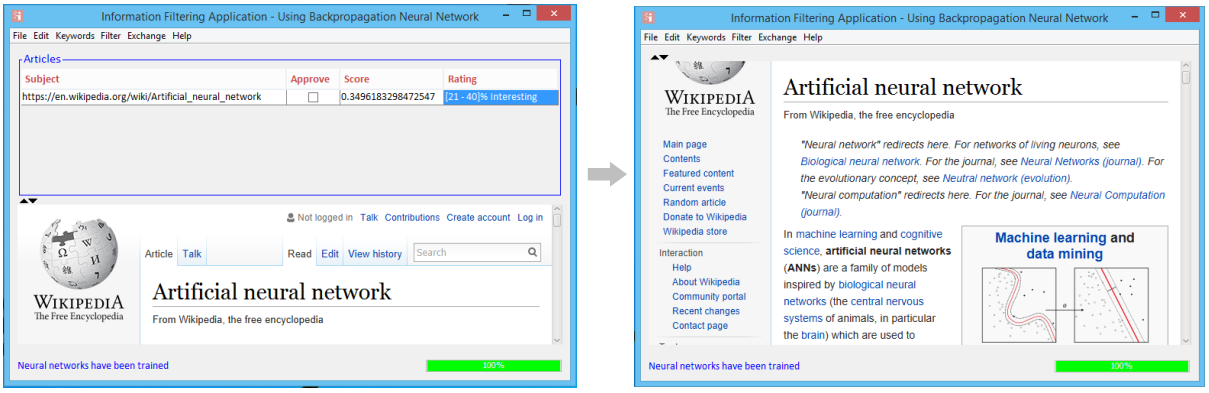
Train Agent's Neural Networks: first place training data file `infofilter.dat` in the same directory where the app was started, then select `Filter > Train Neural Networks`



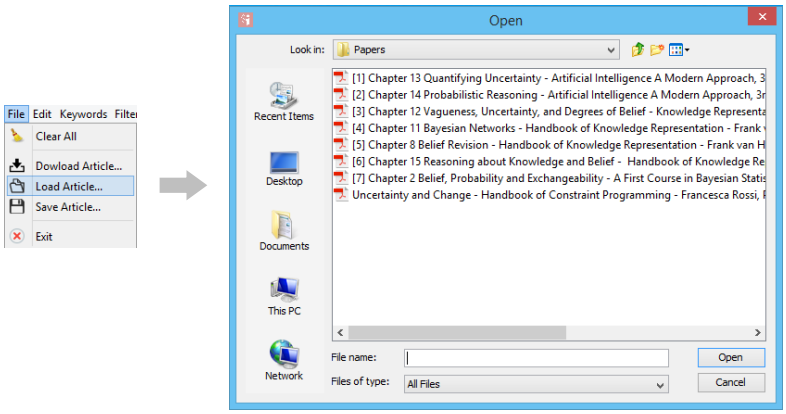
Dowload Articles: select `File > Download Article...`, enter URL of the article to be downloaded, then hit `Download` button.



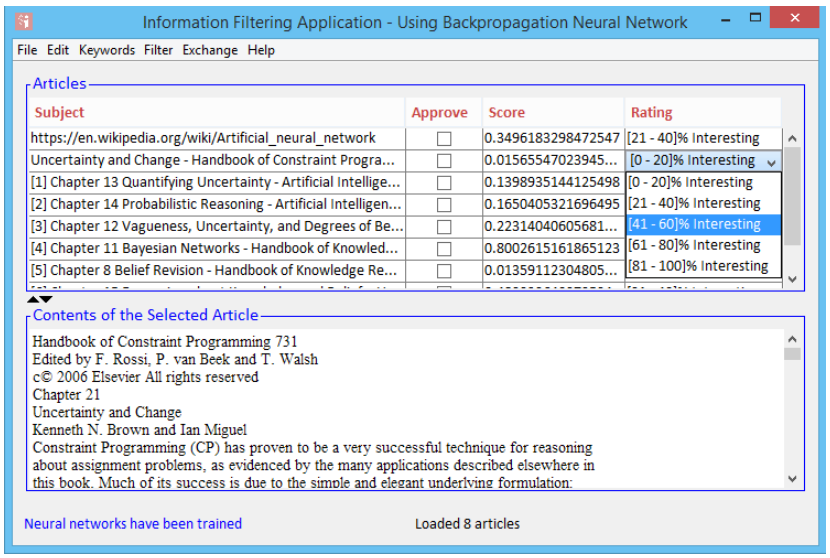
Enlarge Article's Content Viewing Area: click on the `up/down arrow` button in the middle left of the GUI to enlarge the area of viewing article's content.



Load Articles from Personal Computer: select `File > Load Article...`, then on the appearing file chooser dialog select an article file or hold `Shift` key and select multiple files, finally click `Open` button or hit `Enter` on the keyboard.



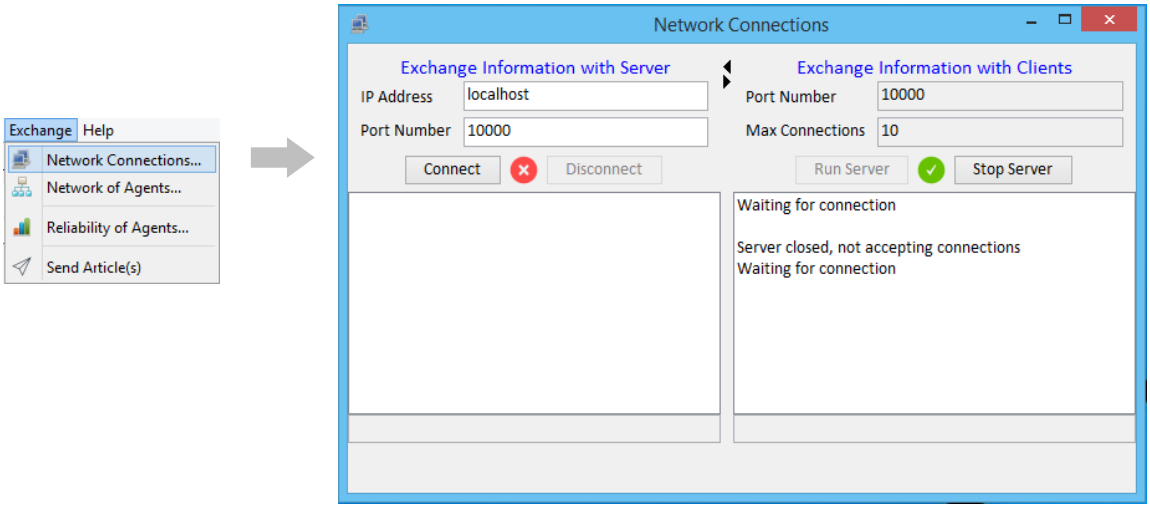
Rate Articles Again: if an article downloaded from the Internet or loaded from PC are assigned wrong `Rating` value, just click on the value and a drop down list will appear to allow selecting other appropriate `rating`.



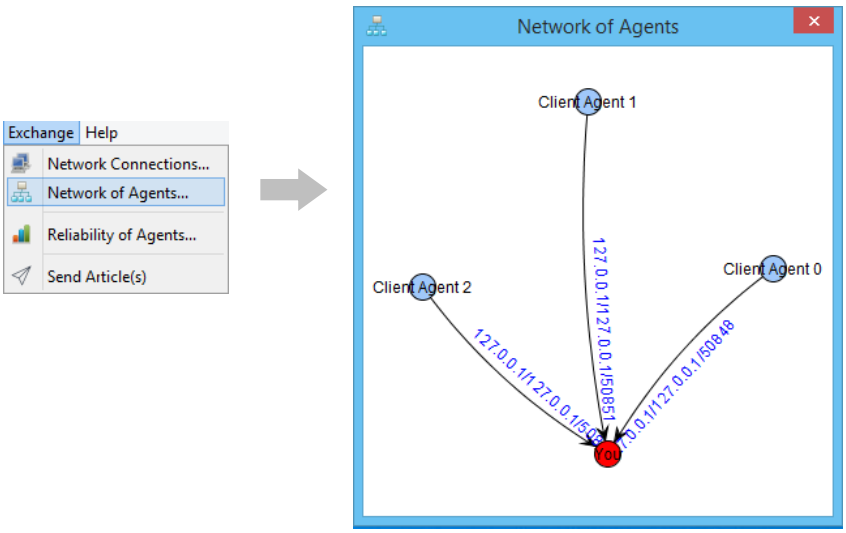
Set up Network Connections: select `Exchange > Network Connections...` to open a dialog.

- If the app plays the role of receiving articles from other apps (other agents), it is a server and on the right hand side of the dialog we just need to specify `Port Number` and `Max Connections` then click `Run Server` button.
- If the app plays the role of sending articles to an other app (lower level agent), it is a client. On the left hand side of the dialog box, specify the `IP Address` and `Port Number` of the server to which it will connect and send articles, then click `Connect` button.

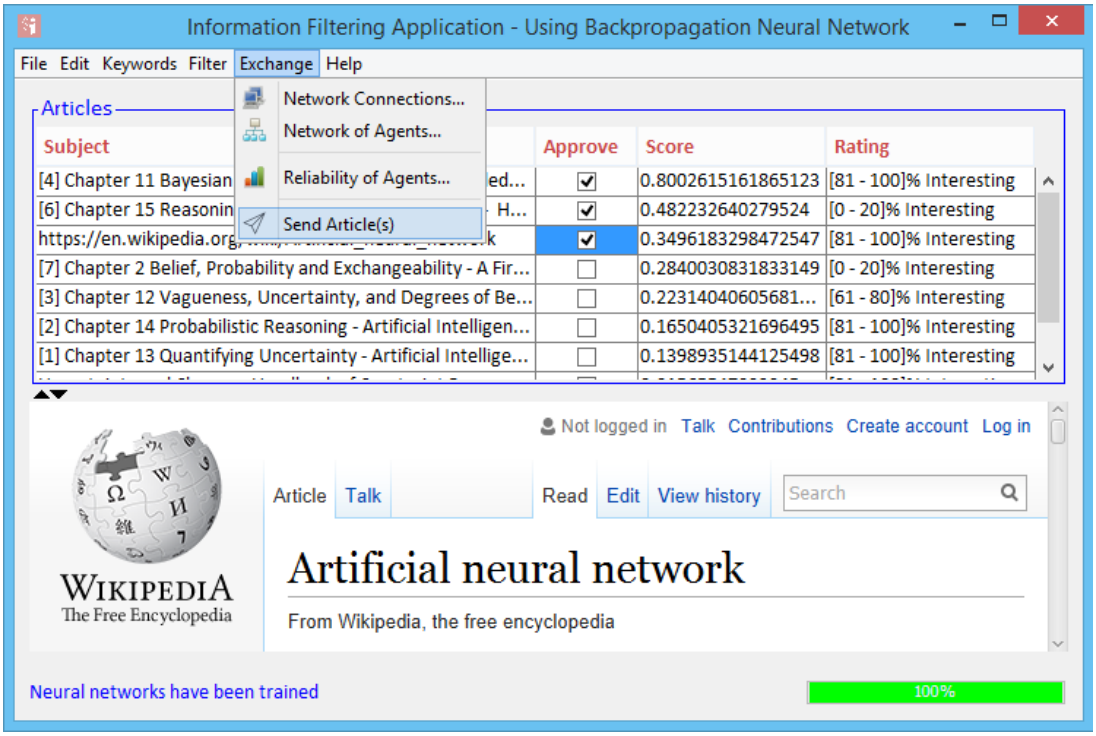
Right below the buttons, there are two text areas for tracing connection status and two text fields for sending messages with app to which this app are connecting. These two text fields are just active when connection is successful.



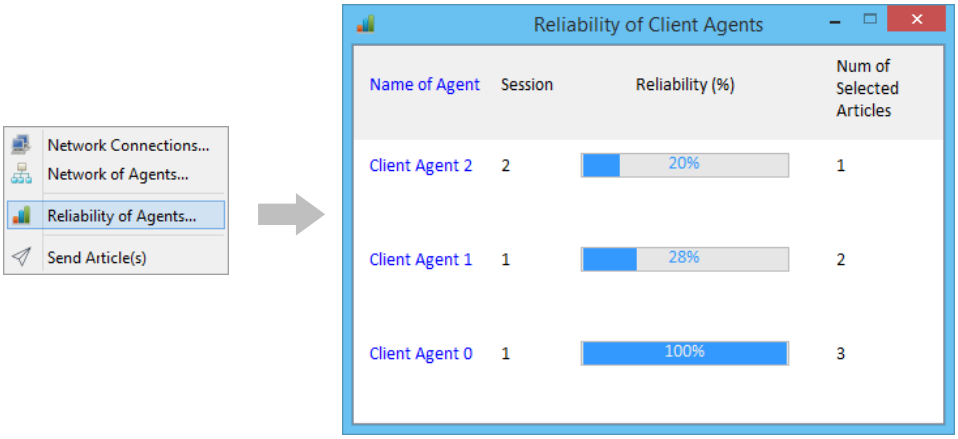
Show All Client Apps: to visualize connections from other apps (from other agents) select `Exchange > Network of Agents...`. A window will appear with a directed graph consisting of nodes representing apps (agents) and edges representing connection directions. Labels of edges are IP addresses and port numbers of client apps.



Send Articles: after connecting to another app (server agent), articles can be sent by fist clicking on the check boxes in the `Approve` column that correspond to articles which has high `score`, then selecting `Exchange > Send Article(s)`.



Check Reliability of Agents: to show reliability of all other apps (client agents) that are connecting to your app directly, select `Exchange > Reliability of Agents...` and a windows will appear with information relating to reliability of these client agents.



And More...: the app has other utility functions such as: just double-clicking on the `Subject` of an article will cause the operating system starts its default application to open the article; or the app can read and score the interestingness of articles in 5 file types, namely `.txt`, `.docx`, `.pptx`, `.pdf` and `.html`.

Bibliography

[1] M. Anthony. *Discrete Mathematics of Neural Networks: Selected Topics*, chapter 1, pages 3–8. Monographs on Discrete Mathematics and Applications. Society for Industrial Mathematics, 2001. ISBN 978-0-89871-480-7.

[2] Y. Bengio and Y. Lecun. *Scaling learning algorithms towards AI*. MIT Press, 2007. URL http://yann.lecun.com/exdb/publis/pdf/bengio-lecun-07.pdf.

[3] L. Bottou. *Large-Scale Machine Learning with Stochastic Gradient Descent*, pages 177–186. Physica-Verlag HD, Heidelberg, 2010. ISBN 978-3-7908-2604-3. doi: 10.1007/978-3-7908-2604-3_16. URL http://dx.doi.org/10.1007/978-3-7908-2604-3_16.

[4] G. R. Brightwell, C. Kenyon, and H. Paugam-Moisy. Multilayer neural networks: One or two hidden layers? In *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*, pages 148–154, 1996. URL http://papers.nips.cc/paper/1239-multilayer-neural-networks-one-or-two-hidden-layers.

[5] J. Callan. Document filtering with inference networks. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '96, pages 262–269, New York, NY, USA, 1996. ACM. ISBN 0-89791-792-8. doi: 10.1145/243199.243273. URL http://doi.acm.org/10.1145/243199.243273.

[6] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[7] L. V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*, chapter 6, pages 289–333. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994. ISBN 0-13-334186-0.

[8] S. Haykin. *Neural Networks and Learning Machines*, chapter 4, pages 112–221. Pearson Education, 3 edition, 2009. ISBN 978-0131471399.

[9] A. K. Jain, J. Mao, and K. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer*, 29:31–44, 1996. doi: 10.1109/2.485891. URL http://dx.doi.org/10.1109/2.485891.

[10] L. C. Jain, M. Seera, C. P. Lim, and P. Balasubramaniam. A review of online learning in supervised neural networks. *Neural Computing and Applications*, 25(3):491–509, 2014. ISSN 1433-3058. doi: 10.1007/s00521-013-1534-4. URL http://dx.doi.org/10.1007/s00521-013-1534-4.

[11] T. John. *Managing the Infoglut: Information Filtering Using Neural Networks*, pages 305–324. Springer US, Boston, MA, 1994. ISBN 978-1-4615-2734-3. doi: 10.1007/978-1-4615-2734-3_16. URL http://dx.doi.org/10.1007/978-1-4615-2734-3_16.

[12] T. Kohonen. *The Basic SOM*, pages 105–176. Springer Berlin Heidelberg, 2001. ISBN 978-3-642-56927-2. doi: 10.1007/978-3-642-56927-2_3. URL http://dx.doi.org/10.1007/978-3-642-56927-2_3.

[13] T. Kohonen. Essentials of the self-organizing map. *Neural Networks*, 37:52 – 65, 2013. ISSN 0893-6080. doi: 10.1016/j.neunet.2012.09.018. URL http://dx.doi.org/10.1016/j.neunet.2012.09.018. Twenty-fifth Anniversay Commemorative Issue.

[14] A. Krogh. What are artificial neural networks? *Nature*, 26:195–197, feb 2008. doi: 10.1038/nbt1386. URL http://dx.doi.org/10.1038/nbt1386.

[15] Q. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Ng. On optimization methods for deep learning. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 265–272, New York, NY, USA, June 2011. ACM. ISBN 978-1-4503-0619-5.

[16] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_2. URL http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

[17] C. P. Lim and R. F. Harrison. An incremental adaptive network for on-line supervised learning and probability estimation. *Neural Networks*, 10(5):925 – 939, 1997. ISSN 0893-6080. URL http://dx.doi.org/10.1016/S0893-6080(96)00123-2.

[18] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL http://dx.doi.org/10.1007/BF02478259.

[19] T. Miyake and S. Miyamoto. Information filtering using fuzzy models. 1:32–37, Oct 1997. doi: 10.1109/ICIPS.1997.672734. URL http://dx.doi.org/10.1109/ICIPS.1997.672734.

[20] S. Miyamoto. Information retrieval based on fuzzy associations. *Fuzzy Sets and Systems*, 38(2):191 – 205, 1990. ISSN 0165-0114. URL http://dx.doi.org/10.1016/0165-0114(90)90149-Z.

[21] T. Nakama. *Comparisons of Single- and Multiple-Hidden-Layer Neural Networks*, pages 270–279. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21105-8. doi: 10.1007/978-3-642-21105-8_32. URL http://dx.doi.org/10.1007/978-3-642-21105-8_32.

[22] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Initial Values of the Adaptive Weights, International Joint Conference of Neural Networks*, pages 21–26, 1990. doi: 10.1109/IJCNN.1990.137819. URL http://dx.doi.org/10.1109/IJCNN.1990.137819.

[23] S. Robertson and H. Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL http://www.staff.city.ac.uk/~sb317/papers/foundations_bm25_review.pdf.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, oct 1986. doi: 10.1038/323533a0. URL http://dx.doi.org/10.1038/323533a0.

[25] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, Nov. 1975. ISSN 0001-0782. doi: 10.1145/361219.361220. URL http://doi.acm.org/10.1145/361219.361220.

[26] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. ISSN 0893-6080. URL http://dx.doi.org/10.1016/j.neunet.2014.09.003.

[27] P. J. Werbos. Computational intelligence from ai to bi to ni. *Proc. SPIE*, 9496: 94960R–94960R–9, 2015. doi: 10.1117/12.2191520. URL http://dx.doi.org/10.1117/12.2191520.

[28] D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003. ISSN 0893-6080. doi: 10.1016/S0893-6080(03)00138-2. URL http://dx.doi.org/10.1016/S0893-6080(03)00138-2.