Rok akademicki 2012/2013

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

PRACA DYPLOMOWA MAGISTERSKA

Aleksy Stanisław Barcz

Implementation aspects of graph neural networks

Opiekun pracy
mgr inż. Zbigniew Szymański

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1988.01.28

Data rozpoczęcia studiów: 2012.02.20

Życiorys

Ukończyłem XXVIII LO im. J.Kochanowskiego w Warszawie, w klasie o profilu matematyczno - informatycznym. Studia inżynierskie ukończyłem w lutym 2012 roku na kierunku Informatyka na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W trakcie studiów I i II stopnia brałem udział w wymianach studenckich programu Athens na Katholieke Universiteit Leuven (*Fundamentals of artificial intelligence*) oraz w Télécom ParisTech (*Emergence in complex systems*).

.....
Podpis studenta

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu .....2013 r

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi Komisji: .....

.....

.....

SUMMARY

This thesis describes the process of implementation of a Graph Neural Network, a classifier capable of classifying data represented as graphs. Parameters affecting the classifier efficiency and the learning process were identified and described. Implementation details affecting the classifier efficiency were described. Important similarities to other connectionist models used for graph processing were highlighted.

Keywords: Graph neural networks, classification, graph processing, recursive neural networks

---

ASPEKTY IMPLEMENTACYJNE GRAFOWYCH SIECI NEURONOWYCH

Praca stanowi raport z samodzielnej implementacji klasyfikatora typu Graph Neural Network (grafowa sieć neuronowa), pozwalającego na klasyfikację danych o strukturze grafowej. W ramach pracy zidentyfikowane zostały istotne dla klasyfikatora parametry, wpływające na przebieg procesu uczenia się klasyfikatora oraz na jakość uzyskanych wyników. Opisane zostały szczegóły implementacyjne klasyfikatora istotne dla jego działania. Klasyfikator został przedstawiony w kontekście podobnych rozwiązań w celu ukazania ścisłych powiązań między istniejącymi modelami przetwarzania danych o strukturze grafowej, opartymi na sieciach neuronowych.

Słowa kluczowe: Grafowe sieci neuronowe, klasyfikacja, przetwarzanie grafów, rekursywne sieci neuronowe

Contents

1. Introduction

The Graph Neural Network model is a connectionist classifier capable of classifying graphs. Most of the other existing neural network-based graph classifiers, such as RAAM [1] or LRAAM [2] and all solutions basing on them are capable of processing certain types of graphs only, in most cases DAGs (directed acyclic graphs) or DPAGs (directed positional acyclic graphs). Several solutions were invented to deal with cyclic graphs, such as introducing a delay in the LRAAM encoding tree [3] or techniques mapping cyclic directed graphs to "recursive equivalent" trees [4]. The problem of nonpositional graphs was also addressed by several authors, either by creating domain-specific encodings used to enforce a defined order on graph nodes [5] or by introducing various modifications to the classifier [6]. However, most of the solutions dealing with cyclic and nonpositional graphs either complicate the classifier model, enlarge the input dataset or (in case of cycles) may result in information loss. The Graph Neural Network model can directly process most types of graphs, including cyclic, acyclic, directed, undirected, positional and nonpositional, which makes it a flexible solution. There is a conceptual similarity between the GNN model and recursive neural networks [7], however, the GNN model adapts a novel learning and backpropagation schema which simplifies processing different types of graphs. This thesis describes the steps of implementation of a GNN classifier, including some details which were not described in the original article [8]. The classifier was implemented in GNU Octave with two ideas in mind: providing a simple interface (similar to that of the Neural Networks toolbox) and maximum flexibility, that is the possibility of processing each kind of data that the theoretical model could deal with. The process of training a GNN and classification results were presented in detail. Listings of most important procedures were included in appendix B.

2. Domains of application

This chapter presents domains where data is organized into a structured form, that is form of sequences or graphs. The necessity of processing differently such kinds of data arose from the structure of the data itself. To present this difference we must first summarize what is the task of classification and regression in the most common sense in data processing domain. A common statistical classifier (which later on is called a *vectorial* classifier) takes as input *samples* from a given *dataset*, representing real world objects, and associates each sample with a category. The samples are fixed-length vectors of numeric values. Each position in such a vector represents a feature of the sample, which is quantified by a real or integer value. The mapping from features to positions in the vector is fixed and must hold for each sample in the dataset. The category is represented by a non-empty fixed-length vector of integer values, where once again the position of each value is meaningful (we can say it's a *positional* representation). For the regression task, a vector of real (or integer) values is associated with each sample instead of a category. The domain of vectorial classifiers is well developed and includes, among other solutions, neural networks, support vector machines, naïve Bayes classifiers and random forests.

In the case of graph processing, the nature of the data is different. Each *sample* is represented by a graph. A *dataset* may consist of a single or several graphs. Each graph consists of nodes, connected with edges. Each node can be described by its *label*, a fixed-length vector of reals. Each edge can also be labelled, with a fixed-length vector of reals of different size. Edges can be directed or undirected. An example of a simple graph was presented in Fig. 2.1.

If such a graph was to be used as input for a common classifier, it would be necessary to transform the structured representation into a plain vector. It could be accomplished in several ways, one of the most obvious would be to perform a preorder walk on the tree and list

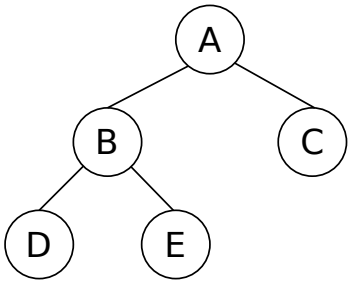

Figure 2.1. A simple binary tree

the node labels in the order they were visited. Such a walk would result in the representation $[A,B,D,E,C]$. It can be seen that the explicit information about node adjacency was lost. Instead, the information is provided implicitly, according to a coding which must be known a priori to properly interpret such a vector representation. That means that a model learning to classify such graph representations would have to learn the encoded relationship, instead of using it from the beginning to learn other, unknown and interesting relationships affecting the samples category. The resulting learning task becomes even harder if such sequential representations contain long-distance relationships. Different encodings from structured data to vectors exist, however, they all share that flaw.

Moreover, the inadequacy of simple vector representation becomes even more apparent when the graph structure becomes more complicated. First of all, if edge labels are present in the vector representation, the representation becomes a mix of data belonging to two different entities - nodes and edges. Once again the classifier doesn't know which part of the data corresponds to the first entity and which to the other. The same applies to directed and undirected edges. If two edges in a graph are connected by a directed edge, presumably one of the nodes in the relation has got a larger impact on the other than vice versa. On the contrary, an undirected or bidirectional edge edge implies an equal impact of both nodes on each other. What if a graph contains both types of edges? How should one type of edges be distinguished from the other one? Secondly, let's consider the case of cyclic dependencies. Even if a meaningful representation of the graph is built ignoring cycles, e.g. by constructing a minimum spanning tree, some explicit information about connections is lost. An example of such data are chemical compounds containing groups of atoms forming cyclic bonds. Another problem lies in the positional nature of vectorial data. If a representation is built by simply storing node labels one after another, this representation becomes vulnerable to any reordering of the children of a node. An additional effort must be made to assure a consistent ordering of the potentially difficult to order data, while the ordering of the children of a node may be irrelevant in the dataset considered.

It can be seen that in order to properly process structured data, a different approach must be used. The data should be processed in a way that exploits properly the information contained in its structure - by means of building a sufficient representation or by processing the structured data directly.

Graph-oriented models based on neural networks were successfully applied in various domains, including chemistry, pattern recognition and natural language processing. In the domain of computer-aided drug design the most important problem to solve is to predict the properties of a molecule prior to synthesizing it. All molecules with a negative prediction can be discarded automatically, reducing the costs of the subsequent laboratory experiments which can focus on the molecules with a positive prediction only [3]. This is the case of

QSAR (quantitative structure-activity relations) and QSPR (quantitative structure-property relations). While traditional processing methods consist of extraction and selection of features from the molecules descriptions, the molecules can be easily represented as undirected graphs and processed with a graph-oriented model [9] [10].

The next domain of interest is document mining. As the amount of XML-formatted information increases rapidly, the problem of determining if a document can be assigned to a given category becomes crucial. As an XML document can be viewed as semi-structured data, graph-processing models can be successfully applied to this task [11]. Another problem focused on document processing is the web page ranking, where documents and the links between them can be described as structured data. A general ranking model can be implemented as a graph-processing model, which allows exploiting page contents and link analysis simultaneously. [12] [8].

In the domain of image processing two pattern recognition tasks can be distinguished. First is the classification of images, either for industrial applications, control and monitoring, or for querying an image database. The second is object localisation, which may be used e.g. for face localisation. For both tasks an image can be represented as a RAG (Region Adjacency Graph), where image segments are represented as nodes and their adjacency is represented as edges which may contain information about e.g. the distances between adjacent segments. For both tasks graph-processing methods can be used, yielding promising results [13] [6] [14].

Another classic example where structure of the data plays a crucial role in its understanding is the natural language processing. In the unconstrained case, the input data may consists of arbitrarily complex sentences. As sentence can be transformed into a graph reflecting its syntax, thus a graph-processing model can be trained to parse such sentences. One of the first graph-processing solutions was already evaluated on such a task [1] and more recent solutions are also present in the literature [15].

3. Graph processing models

A model considered as fully capable of processing structured data, should be able to:

1. build data representation
   a) minimal
   b) exploiting sufficiently the structure of the data
   c) adequate for subsequent processing (classification, regression)
2. perform classification / regression on the structured data
   a) taking into consideration the structure encoded in the representation
   b) with a high generalization capacity

These two main tasks are often intertwined with each other, as a classification procedure may affect the procedure of representation building and vice versa. It is also possible for a model to focus only on representation building, while leaving the task of processing to a common statistical classifier, as support vector machine. Two main families of models capable of processing structured data are the *symbolic* and *connectionist* families. The first one originates in the artificial intelligence domain and focus on inferring relationships by means of inductive logic programming. The *connectionist* models focus on modelling relationships with the use of interconnected networks of simple units. The different models originating from these two families are:

1. inductive logic programming
2. evolutionary algorithms
3. probabilistic models: Bayes networks and Markov random fields
4. graph kernels
5. neural network models

The main area of interest of this thesis are the connectionist models based on neural networks. The connectionist models make the fewest assumptions about the domain of the dataset and thus provide a potentially most general method for processing structured data.

4. History of connectionist models

This chapter summarizes the history of connectionist models used for graph processing. All these models originate from the feed-forward neural networks (FNNs). The history of neural networks begins in 1943 with the McCulloch–Pitts (MCP) neuron model, following with the Rosenblatt perceptron classifier in 1957. The feed-forward neural network model was developed during the following three decades and a conclusive state was reached in all major fields of related research until approximately 1988. The FNN model reached maturity in its field of application: classification and regression performed on unstructured positional samples of fixed size. In the '80s a new branch of the neural networks family began to develop - the recurrent neural networks (RNN). The RNN model is capable of processing sequences of varying length (potentially infinite), which makes them suitable for dealing with time-varying sequences or biological samples of various length [16]. However, a slightly different model had to be invented to properly process graph data.

4.1. Hopfield networks

One of the earliest attempts to classify structured data with neural networks was using the Hopfield networks [3]. A common application of a Hopfield network is an auto-associative memory, which learns to reproduce patterns provided as its inputs. (The task to be learned is the mapping $\boldsymbol{x}_i \Rightarrow \boldsymbol{x}_i$, where $\boldsymbol{x}_i$ is a pattern of fixed size n.) Afterwards, when a new sample is presented to the trained network, the network associates it with the most similar pattern it had learned. Subsequently, it was discovered, that by using a Hopfield network to reproduce a predefined successor of a pattern instead of the pattern itself, the network can be used as an hetero-associative memory, capable of reproducing sequences of patterns ($\boldsymbol{x}_i \Rightarrow \boldsymbol{x}_{i+1}$). The next step towards graph processing was to use Hopfield networks to learn the task of reproducing *all* the successors (or predecessors) of a node, that is to learn the mapping $\boldsymbol{x}_i \Rightarrow succ[\boldsymbol{x}_i]$, where $\boldsymbol{x}_i$ is the ith node label and $succ[\boldsymbol{x}_i]$ denotes a vector obtained by stacking together the labels of all the successors of node $\boldsymbol{x}_i$, one after another. For such task the maximum outdegree of a node (the maximum number of its successors) has to be known prior to the network training. *NIL* patterns are used as extra successors labels whenever a considered node $\boldsymbol{x}_i$ has an outdegree smaller than the maximum value chosen. The last and somehow different application of Hopfield networks was to use a Hopfield network once again as an auto-associative memory, used for retrieving whole graphs. In such case the graph

adjacency matrices ($N \times N$) are encoded into a network having $N(N-1)/2$ neurons [3], where N is the number of nodes in a graph. To obtain an adequate generalisation, graphs isomorphic to the training set are generated and fed to the network [17].

4.2. RAAM

The Recursive Auto-Associative Memory (RAAM) was introduced by Pollack in 1990 [1]. The RAAM model is a generalisation of the Hopfield network model [3], providing means to meaningfully encode directed positional acyclic graphs (DPAGs). A distinctive feature of the RAAM model is that is can be used to encode graphs with labeled terminal nodes only. (The terminal nodes are nodes with outdegree equal to zero, that is nodes having no children. In the case of trees, leaves are terminal nodes.) That is, no node other than the terminal nodes of a graph may be labelled. No edge labels are permitted. The most straightforward domain of application for the RAAM model is thus natural language processing, where sentences can be decomposed to syntax graphs.

The RAAM model is capable of:

— building compressed representation of structured data

— building meaningful representation: similar samples are represented in a similar way

— constrained generalisation: representing data absent in the training set

The RAAM model is composed of two units: *compressor* and *reconstructor*. Together they form a three-layer feed-forward neural network which works as an auto-associative memory. The *compressor* is a fully connected two-layer neural network with n input lines and m output neurons. The number of output neurons, m determines the size of a single encoded node representation. The number of input lines, n must be a multiple of m, such that $n = k \cdot m$, where k is the maximum outdegree of a node in the considered graphs. For each terminal node its representation consists of its original label. For each non-terminal node i its representation, $\boldsymbol{x}_i$ is built by feeding the compressor with encoded representation of the ith nodes children.

To assure that the compressed representation is accurate and lossless, it is fed to the *reconstructor*. The reconstructor is also a fully connected two-layer neural network, however it has m input lines and n output neurons. It is fed with compressed representations of nodes and is expected to produce the original data that was fed to the compressor. This procedure is repeated for all non-terminal nodes of a graph, until all encoded representations can be accurately decoded into original data. More precisely, the representation $\boldsymbol{x}_i$ of the ith node of the graph is given by Eq. 4.1, where f denotes the function implemented by the compressor unit, $\boldsymbol{l}_i$ - the label of ith node, $\boldsymbol{x}_{ch[i]}$ - a vector obtained by stacking representations of all children of ith node one after another, k - the maximum outdegree of a node in the graph.

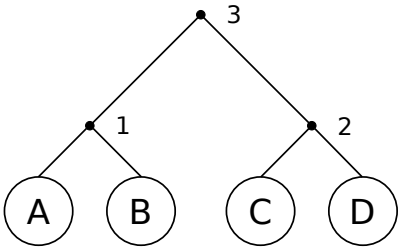

Figure 4.1. A sample graph that can be processed using RAAM

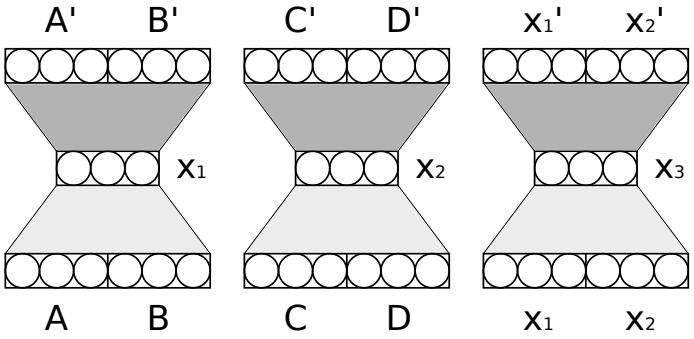

Figure 4.2. Training set for the example graph

$$\boldsymbol{x}_i = \begin{cases} \boldsymbol{l}_i & \text{if } i \text{ is terminal} \\ f(\boldsymbol{x}_{ch[i]}) & \text{otherwise} \end{cases} \tag{4.1}$$

A sample graph that can be encoded using the RAAM model is presented in Fig. 4.1. The non-terminal nodes are enumerated for convenience only (only terminal nodes are labelled).

To encode the sample graph, the representations of nodes 1 and 2 must be built first. The representation of node 1 is built by feeding the pair of labels (A,B) to the compressor which encodes them into the representation $\boldsymbol{x}_1$. The representation $\boldsymbol{x}_1$ is then fed to the reconstructor, which produces a pair of labels (A',B'). If the resulting labels A' and B' are not similar enough to the original labels A and B, the error is backpropagated through the compressor-reconstructor three-layer network. Similarly, the pair (C,D) is processed by the same compressor-reconstructor pair and compressed into the representation $\boldsymbol{x}_2$. Then, the pair $(\boldsymbol{x}_1,\boldsymbol{x}_2)$ is once again fed to the compressor, which produces $\boldsymbol{x}_3$, the representation of the root node. This is also the compressed representation of the whole graph, from which the whole graph can be reconstructed by using the reconstructor unit. The training set, consisting of three label pairs, is presented in Fig. 4.2. The light grey areas denote the compressor network, while the dark grey areas denote the reconstructor. Such a training set (or a larger one if the dataset consists of more than one graph) must be repeatedly processed by the RAAM model in the training phase. When the model is trained, the compression of the whole graph occurs as presented in Fig. 4.3. Reconstruction of the graph is presented in Fig. 4.4. It is worth mentioning that a trained RAAM model can be used to process graphs with different structures.

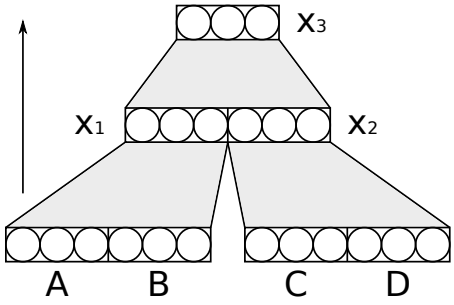

Figure 4.3. Graph compression using trained RAAM model

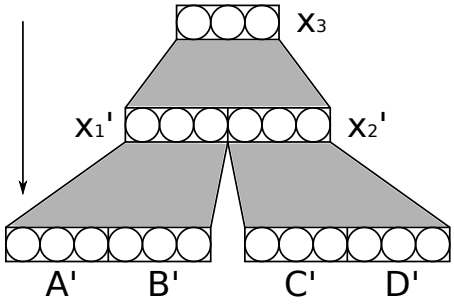

Figure 4.4. Graph reconstruction from $\boldsymbol{x}_3$ using trained RAAM model

A significant feature of the RAAM model is that a small reconstruction error in the case of non-terminal nodes may render the reconstruction of terminal nodes impossible. Therefore in the process of training a RAAM classifier it is necessary to set the acceptable reconstruction error value much smaller for the non-terminal nodes than for the terminal ones.

A major drawback of the RAAM model is the *moving target* problem. That is, a part of the learning set (the representations $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ from the example) changes during training. In such a case the training phase may not converge to an acceptable state [3]. However, a different training schema is possible, similar to the BPTS algorithm [3]. (An extensive description of the BPTS algorithm, *encoding networks*, and the *shared weights* technique is provided in the following chapters.) An encoding network is built out of identical instances of the compressor and reconstructor units (Fig. 4.5), with structure reflecting the structure of the processed graph (if the dataset consists of multiple graphs, such procedure is repeated for every graph in the dataset). All instances of the compressor unit share their weights and all instances of the reconstructor unit share their weights - which is called the *shared weights* technique. The labels of terminal nodes are fed to the processing network and the resulting error can be backpropagated from the last layer using e.g. the Backpropagation Through Structure [18] algorithm (BPTS). It is worth mentioning, that the authors of such modified RAAM model propose using an additional layer of hidden neurons in the compressor and reconstructor units. In such case the light grey and dark grey areas on the figures would denote not only neuron connections between two layers, but also an additional hidden layer. Such modification allows to partially separate the problem of the data model complexity (i.e.

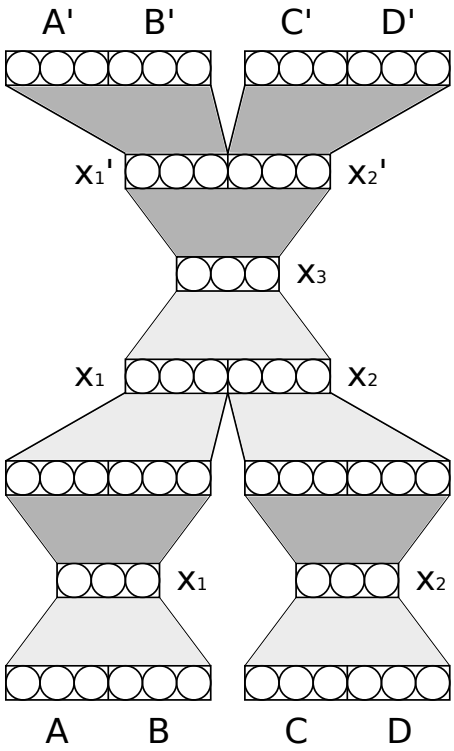

Figure 4.5. RAAM encoding network for the sample graph

how complex should the RAAM model be to properly compress the data) from the size of terminal node labels which affects directly the number of input lines to the compressor unit and thus the compressed representation size.

The most important parameter of the RAAM model is the size of the compressed representation. On one hand, the size should be large enough to contain all the necessary compressed information about the encoded graph. On the other hand, it should be small enough for the compression mechanism to build a minimal representation, which stores only the necessary information about the dataset. If the size is too large, the trained model would store redundant information, memorizing the training set. This would result in a poor ability to process unseen data. Experiments with natural language syntax processing [1] proved that when the size of the compressed representation is accurate for the problem, the RAAM model is showing some constrained generalisation properties. That is, unseen data with structure similar to the training set was processed properly by the trained RAAM model.

A drawback of the standard RAAM model is the termination problem. The reconstructor can't distinguish between terminal representations (node labels) and compressed representations, which should be further reconstructed. To solve this problem, an additional encoding neuron can be introduced (increasing the representation size by one), which takes a different value for terminal and non-terminal representations [19].

4.3. LRAAM

The most important constraint of the RAAM model is the fact that only terminal nodes of the processed graphs (DPAGs) can contain labels. This problem was addressed by the Labeling RAAM model [2] (LRAAM, 1994), which separated the concepts of node labels and node representations. In the RAAM model the terminal nodes are represented by their labels. The LRAAM model introduced the concept of *pointers*, which was used to describe a node representation which has to be learnt, regardless of whether the node is terminal or not. The pointers are built by compressor units (FNN with two or more layers) and they are decoded into graph structure by reconstructor units. More precisely, the pointer to *i*th node of the graph is calculated according to Eq. 4.2, where $\boldsymbol{x}_i$ stands for pointer to the *i*th node, f is the function implemented by the compressor unit, $\boldsymbol{l}_i$ is the *i*th node label, $\boldsymbol{x}_{ch[i]}$ is a vector obtained by stacking pointers to all children of *i*th node one after another and k is the maximum outdegree of a node in the considered graph.

$$\boldsymbol{x}_i = f(\boldsymbol{l}_i,\, \boldsymbol{x}_{ch[i]}) \tag{4.2}$$

Whenever a node outdegree is smaller than k (especially in the case of terminal nodes), the missing child pointers are substituted by the *NIL* pointer, a special value representing the lack of node. The value of the node label is stacked together with all the child pointers values to form an input vector which is fed to the compressor unit. The number of output neurons of the compressor unit (the size of a pointer $\boldsymbol{x}_i$) is m. Let's denote the size of the label $\boldsymbol{l}_i$ by p. The compressor unit must have $n = p + k \cdot m$ input lines which is also the number of reconstructor unit output neurons. The possibility of describing each graph node with its label provides a simple solution to the termination problem [2]. An additional value can be appended to each label, stating if the node is a terminal node or not. By using this method no change in the LRAAM model is needed.

Just like the RAAM model, the LRAAM model experience the problem of the *moving target*. The same technique of *shared weights* can be applied [3], which results in building a large encoding network composed of identical units. A sample graph and the encoding network obtained by cloning the compressor and reconstructor units to reflect the sample graph structure are presented in Fig. 4.6. As A and B are terminal nodes, their labels are fed to the compressor unit altogether with *NIL* pointers representing the missing nodes. Then the compressed representation $\boldsymbol{x}_A$ is built for node A and the compressed representation $\boldsymbol{x}_B$ is built for node B. The representations $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$ are then fed altogether with the label C to the compressor, to build the representation $\boldsymbol{x}_C$ of node C, which is also the compressed representation of the whole graph.

An extension of the LRAAM model exists for cyclic graphs [3]. Whenever an edge forming a cycle is found, it is converted to a single time unit delay (denoted by q^{-1} [7]).

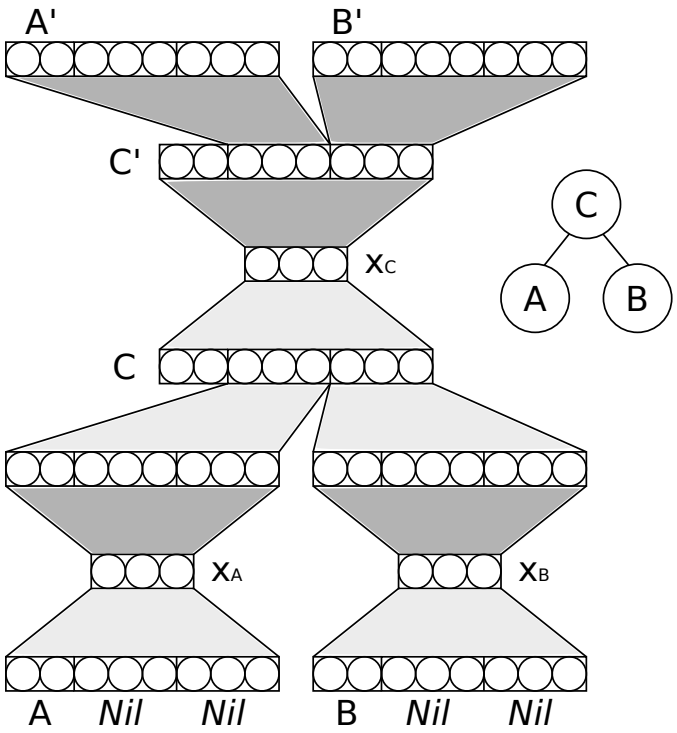

Figure 4.6. LRAAM encoding network for the graph shown

A sample cyclic graph and the resulting LRAAM model with one time delay was presented in Fig. 4.7. The graph presented is similar to the graph used in the previous example, with the exception of the directed edge $A \Rightarrow C$. The additional edge forms a cycle so it must be represented as a time delay. Such an approach makes it possible to deal with cyclic graphs, however it is achieved at the expense of model simplicity. The shared weights technique made it possible to treat the encoding tree structure as a single feed-forward neural network with shared weights. However, after adding time delays the training of the network will have to consist of multiple time steps, repeated until convergence of the pointer values is reached.

A distinctive feature of the LRAAM model is that a compressed representation is built for a given dataset (consisting of DPAGs) and the correctness of the representation is verified by mirroring the compression process and reconstructing the original data. When the obtained representation is accurate enough, the output of the LRAAM model is "frozen" and fed to a separate classifier which processes it and yields classification or regression results. The same applies to any new, unseen data, which is fed to the trained LRAAM model and, if the model was built correctly, is compressed into a meaningful representation. Such separation of representation building and processing can be attractive for two reasons. First, the LRAAM model parameters, such as the size of the representation, can be tuned in a straightforward manner by observing what is the minimal value below which the original data cannot be accurately reconstructed from the compressed vectors. Secondly, any vectorial classifier used for unstructured data can be used to process such compressed representation. On the other hand, it is often safe to presume that not all the data contained in node labels is crucial

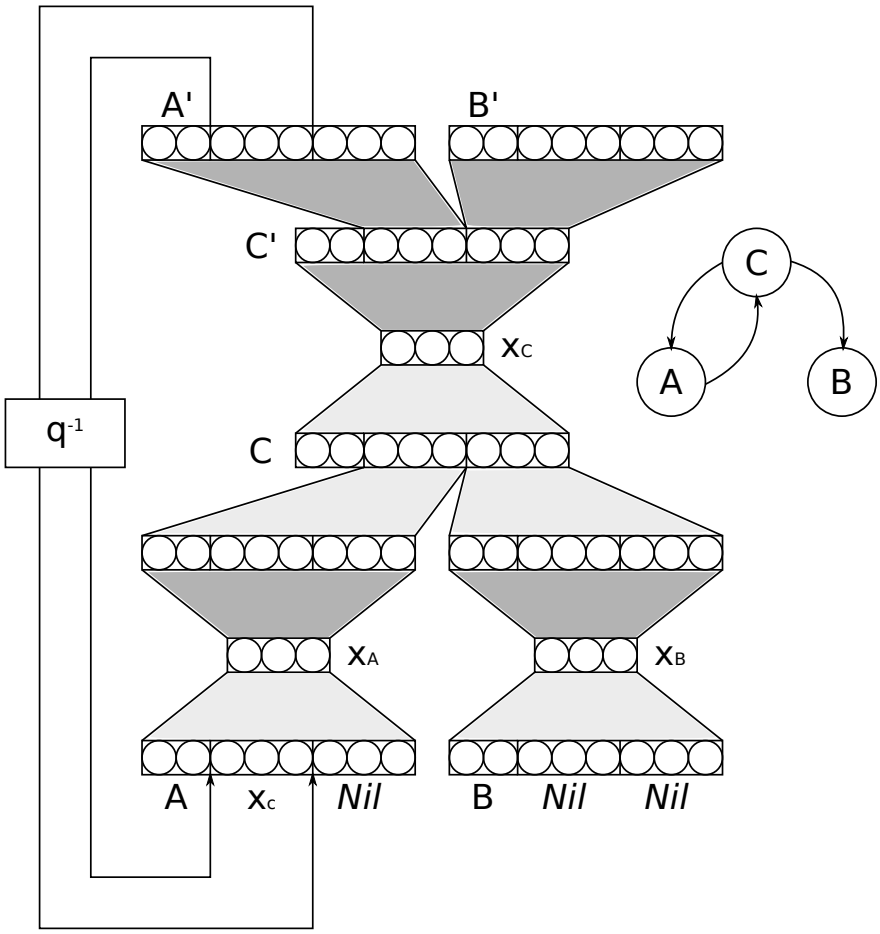

Figure 4.7. LRAAM encoding network for the cyclic graph shown

for the classification/regression process for which the compressed representation is needed. Such an approach lies beyond the standard LRAAM model and was introduced in the *folding architecture* model [18].

4.4. Folding architecture and BPTS

The ideas of *folding architecture* and Backpropagation Through Structure (BPTS) were first introduced in 1996 [20] [18]. The model of the folding architecture is similar to that of LRAAM and is capable of processing rooted DPAGs. (That is DPAGs with a distinguished root node. For each DPAG such a node can be selected.) The folding architecture model is a feed-forward, fully connected multi-layer neural network, consisting of two subsequent parts performing different tasks: the *folding* network and the *transformation* network. The folding network is similar to the LRAAM compressor unit, its input layer consists of $p+k\cdot m$ input lines, p for the processed node label and $k\cdot m$ for the compressed representations of the nodes children. The folding network can consist of any number of sigmoid neuron layers and its last layer produces the compressed representation of a node, of size m. The folding network is applied to every node in the graph, starting from the terminal nodes, so as to provide the internal graph nodes with compressed representations of previous layers nodes. The transformation network is applied to the root node only. It can consist of any number of sigmoid neuron layers and an output layer. It takes as input the compressed representation of the root node and produces an output, which should match the expected output for a graph. Therefore, the transformation network is used to perform classification or regression tasks in terms of whole graphs. Let's denote the folding network as f and the transformation network as g. The function f can be described by Eq. 4.3, where $\boldsymbol{x}_i$ stands for the ith node representation, $\boldsymbol{l}_i$ is the ith node label and $\boldsymbol{x}_{ch[i]}$ is a vector obtained by stacking representations of all children of ith node one after another. The g function can be described by Eq. 4.4, where $\boldsymbol{o}_r$ is the output of the root node.

$$\boldsymbol{x}_i = f(\boldsymbol{l}_i,\, \boldsymbol{x}_{ch[i]}) \tag{4.3}$$

$$\boldsymbol{o}_r = g(\boldsymbol{x}_r) \tag{4.4}$$

The original idea behind the *folding architecture* is that the compressed representation is built only for classification purpose and is fed directly to the transformation network. The output of the transformation network is then compared with the expected output and the error can be backpropagated through the folding architecture network by using a gradient-descent procedure, Backpropagation Through Structure. BPTS was invented as a generalisation of the Backpropagation Through Time method (BPTT [21]), which in turn was invented for

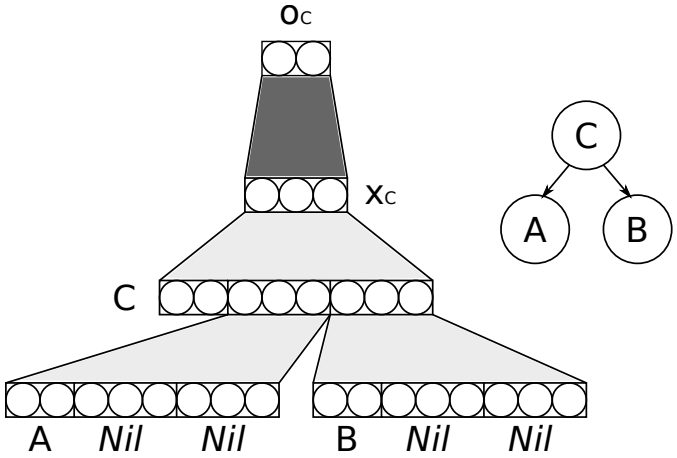

Figure 4.8. Virtual unfolding, reflecting the sample graph structure

error backpropagation in recurrent neural networks. BPTS can be described in terms of the *unfolded* network. The unfolded network is never built physically but can be imagined as a graph built of folding network instances in a way which reflects the structure of the processed graph, with the transformation network added on top of it (attached to the representation of the root node). An unfolding network for a sample graph is presented in Fig. 4.8. The light grey areas are instances of the folding network, while the dark grey area is the transformation network, which for the root node C produces output $\boldsymbol{o}_C$.

To explain the idea of BPTS it is necessary to briefly summarize the idea of BPTT (a detailed explanation can be found in RNN-concerned publications, e.g. [22]). Let's consider a fully-connected recurrent neural network, designed for classifying sequences of samples of size m. The network consists of a single layer of n units with $n \times n$ recurrent connections, producing an output $\boldsymbol{y}(t)$ at time t. Let $\boldsymbol{x}^s(t)$ denote the m-tuple of input signals corresponding to the sample fed at time t. Further, let $\boldsymbol{x}(t)$ be the merged input fed to the network at time t, obtained by concatenating the vectors $\boldsymbol{x}^s(t)$ and $\boldsymbol{y}(t)$. To distinguish between the elements of vector $\boldsymbol{x}(t)$ corresponding to $\boldsymbol{x}^s(t)$ and to $\boldsymbol{y}(t)$, let's introduce two subsets of indices: I and U (Eq. 4.5).

$$x_j(t) = \begin{cases} x_j^s(t) & \text{if } j \in I \\ y_j(t) & \text{if } j \in U \end{cases} \tag{4.5}$$

Let w_{kj} denote the network weight on connection to the kth neuron from input x_j, $net_k(t)$ denote the weighted sum of neuron inputs fed to the activation function of the kth neuron, f_k (Eq. 4.6) and $J(t)$ denote the overall mean square error of the network at time t.

$$net_k = \sum_{j \in (I \cup U)} w_{kj} x_j \tag{4.6}$$

$$y_k = f_k(net_k) \tag{4.7}$$

$$J(t) = -\frac{1}{2}\sum_{k\in U}[e_k(t)]^2 \tag{4.8}$$

$$e_k(t) = d_k(t) - y_k(t) \tag{4.9}$$

Let's consider a recurrent network which was operating from a starting time t_0 up to time t. We may represent the computation process performed by the network by *unrolling* the network in time, that is building a feed-forward neural network made of identical instances of the considered recurrent neural network, one instance per time step τ, $\tau \in (t_0, t]$. To compute the gradient of $J(t)$ at time t it is necessary to compute the error values $\varepsilon_k(\tau)$ and $\delta_k(\tau)$ for $k \in U$ and $\tau \in (t_0, t]$ by means of equations 4.10, 4.11 and 4.12.

$$\varepsilon_k(t) = e_k(t) \tag{4.10}$$

$$\delta_k(\tau) = f'_k(net_k(\tau))\varepsilon_k(\tau) \tag{4.11}$$

$$\varepsilon_k(\tau - 1) = \sum_{j\in U} w_{jk}\delta_j(\tau) \tag{4.12}$$

Then, the gradient of $J(t)$ is calculated with respect to each weight w_{ij} by the means of equation 4.13.

$$\frac{\partial J(t)}{\partial w_{ij}} = \sum_{\tau=t_0+1}^{t} \delta_i(\tau)x_j(\tau - 1) \tag{4.13}$$

At time t an external error $\boldsymbol{e}(t)$ is *injected* to the network, usually being the difference between the trained network output at time t: $\boldsymbol{y}(t)$ and the expected output $\boldsymbol{d}(t)$ (Eq. 4.9). The subsequent steps compute the error $\varepsilon(\tau)$ by backpropagating the original error through the layers of the unrolled neural network.

The BPTS method implements the BPTT algorithm. Backpropagation starts at the last layer of the virtual unfolding network, where the classification/regression error is calculated (the last layer of the transformation network applied to the root node). The error is injected to this layer and backpropagated using the BPTT algorithm down to the first layer of the folding network applied to the root node. The error is then backpropagated to the last layers of the folding network applied to the roots children, as if there was a physical connection. Such backpropagation continues down to the first layers of folding network applied to the terminal nodes.

The folding architecture model introduced new important ideas in the domain of connectionist graph processing models. First of all, the representation building model is simpler than the LRAAM model and the folding architecture converges much faster than LRAAM

for the same datasets [20]. Secondly, three important concepts were adapted from the domain of recurrent neural networks: the error injection, BPTS and the unfolding of the network as a generalisation of unrolling a recursive neural network.

4.5. Generalised recursive neuron

The *generalised recursive neuron*, introduced in 1997 [23] is a generalisation of the recurrent neuron, which in turn is used in recurrent neural networks (RNNs). It was created to provide an elementary component for the graph processing models which would be by definition better suited to solve graph processing tasks than the standard neuron. It is beyond the scope of this thesis to describe in detail the idea itself and its applications. Nevertheless this summary of the connectionist models would be incomplete without mentioning the generalised recursive neuron, as it was used in some of the following models instead of the common neural network neuron, yielding promising results [7].

A generalised recursive neuron has two kinds of inputs:

— plain neuron inputs, which are fed with elements of the currently processed node label

— recursive inputs, which are fed with the memorized output of the neuron for all children nodes of the currently processed node

In such a way, the neuron output changes after each training algorithm iteration, according to its output for all the children nodes.

4.6. Recursive neural networks

The folding architecture model had a large impact on a theory introduced in 1998, the *structural transduction* formalism [7]. It is beyond the scope of this thesis to describe the formalism itself, let's mention however its aspects that highly affected the subsequent connectionist models described. The *structural transduction*, in general, is a relation which maps a labelled DPAG (only node labels, no edge labels) into another DPAG. The type of transductions that the authors focus on are *IO-isomorph* transductions, that is transductions that don't change the graph topology, only the node labels. (According to the authors managing transductions which are not *IO-isomorph* is highly nontrivial and is an open research problem.) Let's describe an *IO-isomorph* transduction. Let G_s be the original DPAG with labelled nodes. Let G_o be a graph with same topology that G_s but with node labels replaced by expected node outputs for each node (it's a node classification/regression problem). A transduction $G_s \Rightarrow G_o$ can be described in terms of two functions, f and g, where f is the *state transition function*, which builds the representation (the *state*) $\boldsymbol{x}$ of the graph G_s for the model and g is the *output function*, which produces the expected output graph G_o according to the representation $\boldsymbol{x}$ and the original graph G_s. More precisely, for

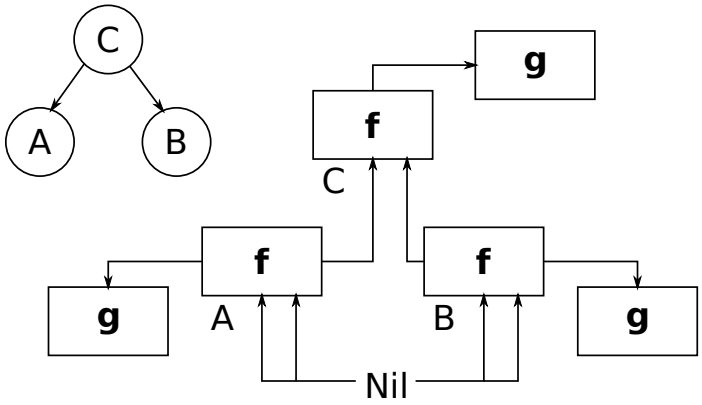

Figure 4.9. A sample acyclic graph and the corresponding encoding network

each node i belonging to the original graph G_s its state $\boldsymbol{x}_i$ and output $\boldsymbol{o}_i$ are defined by Eq. 4.14 and 4.15, where $\boldsymbol{l}_i$ is the ith node label and $\boldsymbol{x}_{ch[i]}$ is a vector obtained by stacking representations of all children of ith node one after another. In the original equations the ith node itself was also an argument of both functions, however, it was unnecessary from the point of view of recursive neural networks and therefore was omitted.

$$\boldsymbol{x}_i = f(\boldsymbol{l}_i, \boldsymbol{x}_{ch[i]}) \tag{4.14}$$

$$\boldsymbol{o}_i = g(\boldsymbol{l}_i, \boldsymbol{x}_i) \tag{4.15}$$

The transduction can be implemented e.g. by a hidden recursive model (HRM) or by a *recursive neural network* (a generalisation of a recurrent neural network, which is able to process not only sequences, but also DPAGs). In the case of a recursive neural network the functions f and g are implemented by two feed-forward neural networks. Identical instances of the f network are connected according to the G_s graph structure, creating the *encoding network*. (The encoding network is the recursive neural network *unfolded through the structure* of the given DPAG.) Calculation of the state $\boldsymbol{x}$ is performed by applying the f network to the terminal nodes of G_s and then proceeding up to the root, according to the encoding network topology. When the state calculation is finished, the g network is applied to every node state $\boldsymbol{x}_i$, producing the requested output $\boldsymbol{o}_i$. A sample graph and the corresponding encoding network are presented in Fig. 4.9.

As various kinds of neural network can be used as the f function (first, second-order recurrent networks etc.), the recursive neural network model can become really complex and computationally powerful. However, for the scope of this work, it is sufficient to mention that the concepts of *state transition function*, *output function* and *encoding network unfolded through structure* (originating from the *folding architecture*) were later reused in the Graph Neural Network model. Another important feature of this model is a successful adaptation

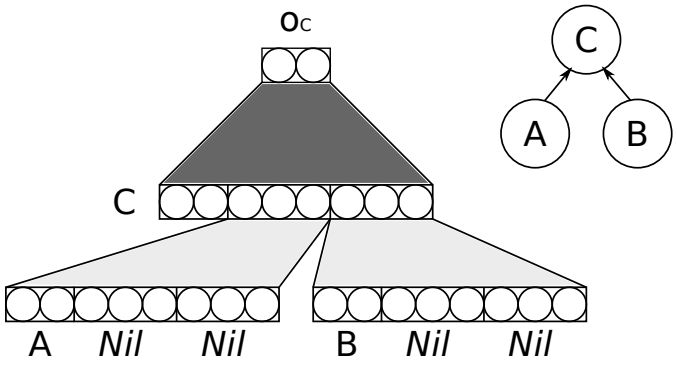

Figure 4.10. A sample acyclic graph and the corresponding encoding network

of ideas originating from recurrent neural network domain, just like in case of the *folding architecture*. Authors of both models suggested that adaptation of other RNN ideas may also be possible, which could lead to novel graph processing solutions [18] [7].

4.7. Graph machines

In 2005, another model for DPAG processing was proposed, the Graph Machines [10]. The model is similar to that of *folding architecture* and *recursive neural networks*. A function f is evaluated at each node of a graph except the root node. Then, the output function g, which can be the same as f, is evaluated at the root node, and yields an output for the whole graph. The function f can be described by Eq. 4.16, where $\boldsymbol{x}_i$ stands for the ith node representation, $\boldsymbol{l}_i$ is the ith node label and $\boldsymbol{x}_{pa[i]}$ is a vector obtained by stacking representations of all parents of ith node one after another. The g function can be described by Eq. 4.17, where $\boldsymbol{o}_r$ is the output of the root node.

$$\boldsymbol{x}_i = f(\boldsymbol{l}_i, \boldsymbol{x}_{pa[i]}) \tag{4.16}$$

$$\boldsymbol{o}_r = g(\boldsymbol{l}_r, \boldsymbol{x}_{pa[r]}) \tag{4.17}$$

The instances of the f function are connected together to form an *encoding network* for a given graph, which reflects the structure of the graph. Such encoding network is built for every graph in the training set. The resulting set of encoding networks is called a *graph machine*. A single encoding network and the corresponding sample graph are shown in Fig. 4.10, where the light grey areas denote instances of the f network and the dark grey area denote the g network.

The instances of f function share weights among the instances belonging to single encoding network, and among all the encoding networks. The instances of g function share weights among all the encoding networks. (This is called the *shared weights* technique.) The f and g functions can be implemented by neural networks. Their training occurs on all

the graphs from the training set simultaneously. That is, $\frac{\partial J}{\partial \boldsymbol{w}}$ is summed over all function instances among all the graphs, where $\boldsymbol{w}$ is the set of all weights belonging either to f or g function. A standard gradient optimization algorithm can be used for the training phase (Levenberg-Marquardt, BFGS, etc.).

The authors of graph machines stated explicitly two fundamental rules which were before implicitly applied in various graph processing connectionist models:

— *The structure of a model should reflect the structure of the processed graph.*

— *The representation of structured data should be learnt instead of being handcrafted. [3]*

5. Graph neural network implementation

The Graph Neural Network model [8] (GNN) is a quite recent (2009) connectionist model, based on recursive neural networks and capable of classifying almost all types of graphs. The main difference between the GNN model and previous connectionist models is the possibility of processing directly nonpositional and cyclic graphs, containing both node labels and edge labels. Although some similar solutions were introduced in an earlier model, the *RNN-LE* [6] in 2005, it was the GNN model that combined several techniques with a novel learning schema to provide a direct and flexible method for graph processing.

5.1. Data

The GNN model is built once for a training set of graphs. In fact, a whole set of graphs can be merged into one large disconnected graph, which can then be fed to the model. For a given dataset each node n is described by a node label $\boldsymbol{l}_n$ of fixed size $|\boldsymbol{l}_n| \geq 1$. Each directed edge $u \Rightarrow n$ (from node u to node n) is described by an edge label $\boldsymbol{l}_{(n,u)}$ of fixed size $|\boldsymbol{l}_{(n,u)}| \geq 0$. To deal with both directed and undirected edges, the authors propose to include a value d_l in each edge label, denoting the direction of an edge. However, for a maximally general model, in this implementation all the edges were considered as directed. Undirected edges were encoded prior to processing as pairs of directed edges with same labels.

A GNN model can deal with both *graph-focused* and *node-focused* tasks. In graph-focused tasks, for each graph an output $\boldsymbol{o}_n$ of fixed size $|\boldsymbol{o}_n| \geq 1$ is sought, which can denote e.g. the class of the graph. In the domain of chemistry, where each graph describes a chemical compound, this could describe e.g. the reactivity of a compound. In node-focused tasks, such output $\boldsymbol{o}_n$ is sought for every node in every graph. An example of such task can be the face localisation problem, where for each region of an image the classifier should determine, if it is a part of the face or not. In the rest of this thesis, the *node-focused* task is described, unless stated otherwise.

For this implementation each graph was represented by three .csv files. Each ith row of the *nodes file* contained the ith node label (comma-separated). Each row of the *edges file* contained information about a directed edge $u \Rightarrow n$: the u node index (row number in nodes file), the n node index and the edge label (comma-separated). Each ith row of the *outputs file* contained the ith node output (comma-separated). An example is provided in Appendix A.

5.2. Computation units

The GNN model consists of two computation units, $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, where the $_{\boldsymbol{w}}$ subscript denotes the fact that both units are functions parametrized by a vector of parameters $\boldsymbol{w}$, which is separate for the f and for the g function. The $f_{\boldsymbol{w}}$ unit is used for building representation (the *state*) $\boldsymbol{x}_n$ of a single node n. The $g_{\boldsymbol{w}}$ unit is used for producing output $\boldsymbol{o}_n$ for a node n, basing on its representation $\boldsymbol{x}_n$. For a graph-focused task, the representation of the root node is fed to the $g_{\boldsymbol{w}}$ function to produce an output $\boldsymbol{o}_g$ for the whole graph. It is important to remind, that for a given classifier there is only one $f_{\boldsymbol{w}}$ unit and one $g_{\boldsymbol{w}}$ unit (like in the recursive neural network model). All instances of the $f_{\boldsymbol{w}}$ unit share their weights and all instances of the $g_{\boldsymbol{w}}$ unit share their weights.

Let's denote by $ne[n]$ the neighbors of node n, that is such nodes u that are connected to node n by a directed edge $u \Rightarrow n$. Let's further denote by $co[n]$ the set of directed edges pointing from $ne[n]$ towards node n (edges $u \Rightarrow n$). The general forms of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ functions are defined by equations Eq. 5.1 and Eq. 5.2, where $\boldsymbol{l}_n$ denotes the nth node label, $\boldsymbol{l}_{co[n]}$ denotes the set of edge labels from $co[n]$, $\boldsymbol{x}_{ne[n]}$ denotes *states* of nodes from $ne[n]$, and $\boldsymbol{l}_{ne[n]}$ denotes their labels.

$$\boldsymbol{x}_n = f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{co[n]}, \boldsymbol{x}_{ne[n]}, \boldsymbol{l}_{ne[n]}) \tag{5.1}$$

$$\boldsymbol{o}_n = g_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{l}_n) \tag{5.2}$$

For this implementation, minimal forms of these definitions were chosen:

$$\boldsymbol{x}_n = f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{co[n]}, \boldsymbol{x}_{ne[n]}) \tag{5.3}$$

$$\boldsymbol{o}_n = g_{\boldsymbol{w}}(\boldsymbol{x}_n) \tag{5.4}$$

These forms were chosen to prove that the model is capable of building a sufficient representation of each node. That is, the model should be able to encode all the necessary information from a node label $\boldsymbol{l}_n$ into the state $\boldsymbol{x}_n$. This approach proved later to be successful.

From two forms of the $f_{\boldsymbol{w}}$ function mentioned in the original article [8], the *nonpositional form* was chosen. The reason behind this choice was to provide the model with the most general function possible, which could deal with both positional and nonpositional graphs. The nonpositional form was also the one yielding better results in the experiments conducted by the authors [8]. The final definitions of the $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ functions are shown below. All instances of the $h_{\boldsymbol{w}}$ unit share their weights.

$$\boldsymbol{x}_n = \sum_{u \in ne[n]} h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u) \tag{5.5}$$

$$\boldsymbol{o}_n = g_{\boldsymbol{w}}(\boldsymbol{x}_n) \tag{5.6}$$

The units $h_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ were implemented as fully-connected three layer feed-forward neural networks (input lines performing no computation, a single hidden layer and an output layer). For both units the hidden layer consisted of *tanh* neurons. For the $h_{\boldsymbol{w}}$ unit the output layer consisted of *tanh* neurons. That's because the output of the $h_{\boldsymbol{w}}$ unit contributes to the state value and therefore should consist of bounded values only. For the $g_{\boldsymbol{w}}$ unit the output layer could consist either of *tanh* or linear neurons, depending on the values of $\boldsymbol{o}_n$ that were to be learned.

At this point it's worth mentioning that the final value of $\boldsymbol{x}_n$ is calculated as a simple sum of $h_{\boldsymbol{w}}$ outputs. This corresponds to a situation where all the $h_{\boldsymbol{w}}$ output values are passed to a neural network in which the set of weights corresponding to a single $h_{\boldsymbol{w}}$ input are shared amongst all the $h_{\boldsymbol{w}}$ inputs. If we consider that a three layer FNN used as $h_{\boldsymbol{w}}$ unit is already an universal approximator, the use of such an additional neural network which just sums all $h_{\boldsymbol{w}}$ values using the same shared set of weights is unnecessary. A simple sum should be sufficient and experimental results showed that this assumption stands.

The $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ units are presented in Fig. 5.1 and Fig. 5.2, where the comma-separated list of inputs stands for a vector obtained by stacking all the listed values one after another.

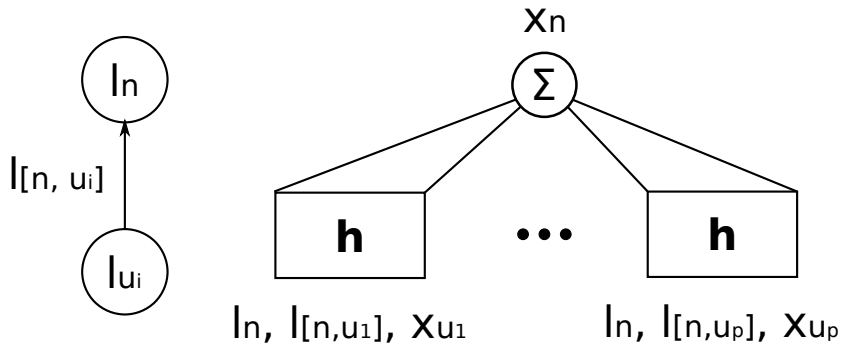

Figure 5.1. The $f_{\boldsymbol{w}}$ unit for a single node and one of the corresponding edges

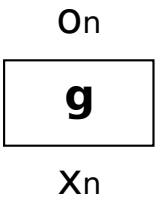

Figure 5.2. The $g_{\boldsymbol{w}}$ unit for a single node

The weights of both the $h_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ units were initialised according to the standard neural network practice, to avoid saturation of any *tanh* activation function: $net_j = \sum_i w_{ji} y_i \in (-1, 1)$, where net_j is the weighted input to jth neuron, y_i is the ith input value and w_{ji} is the weight corresponding to the ith input. The initial input weights of the $g_{\boldsymbol{w}}$ unit were divided by an additional factor, i.e. the maximum node indegree of the processed graphs, to take into consideration the fact, that the input of $g_{\boldsymbol{w}}$ unit consists of a sum of $h_{\boldsymbol{w}}$ outputs. All the input data (node and edge labels) was normalised appropriately before feeding to the model.

5.3. Encoding network

Graph processing by a GNN model consists of two steps: building representation $\boldsymbol{x}_n$ for each node and producing an output $\boldsymbol{o}_n$. As the representation of a single node depends on other nodes representations, an encoding network for every graph is built, reflecting the structure of the graph. The encoding network consists of instances of the $f_{\boldsymbol{w}}$ unit connected according to the graph structure with a $g_{\boldsymbol{w}}$ unit attached to every $f_{\boldsymbol{w}}$ unit. A sample graph and its encoding network are presented in Fig. 5.3. It can be seen that, as a cyclic dependence exists in the sample graph, the calculation of the node *states* should be iterative, until at some point convergence is reached.

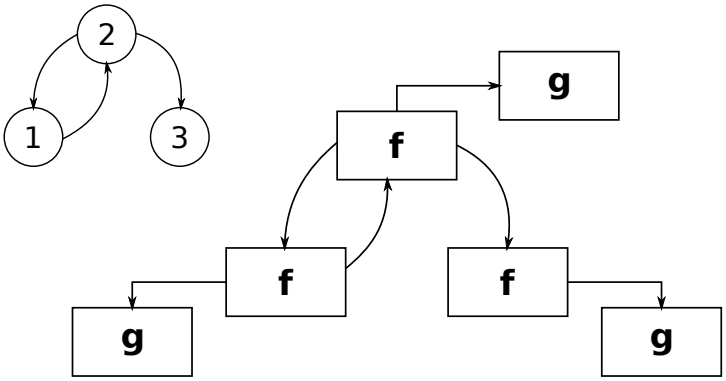

Figure 5.3. A sample graph and the corresponding encoding network

5.4. General training algorithm

Let's denote by $\boldsymbol{x}$ the *global state* of the graph, that is the set of all $\boldsymbol{x}_n$ for every node n in the graph. Let's denote by $\boldsymbol{l}$ and $\boldsymbol{o}$ the sets of all labels and all outputs, respectively. Let's further denote by $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ (*global transition function* and *global output function*) the stacked versions of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ functions, respectively. Now, equations 5.3 and 5.4 can be rewritten as Eq. 5.7 and Eq. 5.8.

$$\boldsymbol{x} = F_{\boldsymbol{w}}(\boldsymbol{l},\ \boldsymbol{x}) \tag{5.7}$$

$$\boldsymbol{o} = G_{\boldsymbol{w}}(\boldsymbol{x}) \tag{5.8}$$

The GNN training algorithm can be described as follows:

1. initialize $h_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ weights
2. until stop criterion is satisfied:
   a) initialize $\boldsymbol{x}$ randomly
   b) FORWARD: calculate $\boldsymbol{x} = F_{\boldsymbol{w}}(\boldsymbol{l},\ \boldsymbol{x})$ until convergence
   c) BACKWARD: calculate $\boldsymbol{o} = G_{\boldsymbol{w}}(\boldsymbol{x})$ and backpropagate the error
   d) update $h_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ weights

The stop criterion used in this implementation was a maximum number of iterations.

5.5. Unfolded network and backpropagation

To solve the problem of cyclic dependencies, the GNN models adapts a novel learning algorithm for the encoding network. The encoding network is virtually *unfolded* through time until at time t_m the state $\boldsymbol{x}$ converges to a *fixed point* $\hat{\boldsymbol{x}}$ of the function $F_{\boldsymbol{w}}$. Then the output $\boldsymbol{o}$ is calculated. Such unfolded network for the sample graph is presented in Fig. 5.4. Each time step consists of evaluating the $f_{\boldsymbol{w}}$ function at every node. What is important, the connections between nodes are taken into consideration only between time steps. In such a way, the problem of cycles ceases to exist and the processed graph can be even fully connected.

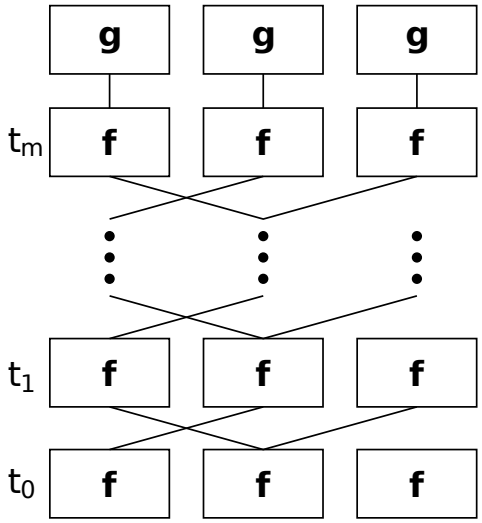

Figure 5.4. Unfolded encoding network for the sample graph

After the output $\boldsymbol{o}$ is calculated, the error $e_n = (d_n - o_n)^2$ is injected to the corresponding $g_{\boldsymbol{w}}$ unit for every node n, where d_n denotes the expected node output. The error is backpropagated through the $g_{\boldsymbol{w}}$ layer, yielding the value $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial x}(\hat{\boldsymbol{x}})$. That value is backpropagated through the unfolded network using the BPTT/BPTS algorithm. Additionally, at each time step the $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial x}(\hat{\boldsymbol{x}})$ error is injected to the $f_{\boldsymbol{w}}$ layer, as presented in Fig. 5.5. In such a way the error backpropagated through the $f_{\boldsymbol{w}}$ layer at time t_i comes from two sources. First, it is the output error of the network $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial x}(\hat{\boldsymbol{x}})$. Secondly, it is the error backpropagated from the subsequent time layers of the $f_{\boldsymbol{w}}$ unit from all nodes connected with the given node u by an edge $u \Rightarrow n$.

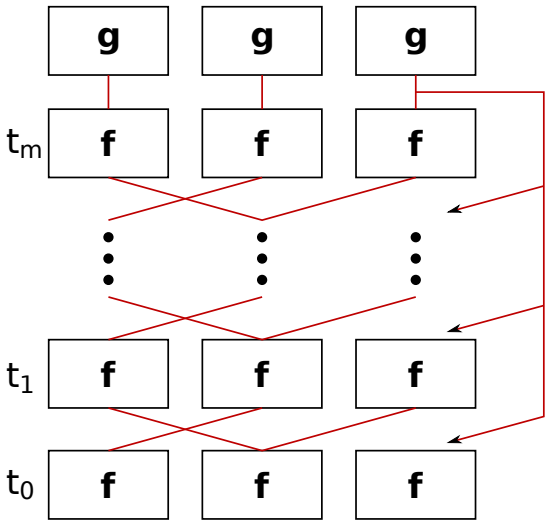

Figure 5.5. Error backpropagation through the unfolded network

By injecting the same error $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial x}(\hat{\boldsymbol{x}})$ at each time step an important assumption is made, which leads to a simplification of the whole backpropagation process. If the state $\boldsymbol{x}$ converged to a *fixed point* $\hat{\boldsymbol{x}}$ of function $F_{\boldsymbol{w}}$ at time t_m, then it can be safely assumed that using the value $\hat{\boldsymbol{x}}$ at every previous time step t_i instead of $\boldsymbol{x}(t_i)$ would yield the same result at time t_m.

Storing the intermediate values of $\boldsymbol{x}(t_i)$ and backpropagating the error directly using the BPTT/BPTS algorithm would be memory consuming. However, due to the assumption that $\boldsymbol{x}(t_i) = \hat{\boldsymbol{x}}$, a different backpropagation algorithm can be used [8], originating from the domain of recurrent neural networks: the Almeida-Pineda algorithm [21] [22].

Basically, the modified Almeida-Pineda algorithm consists of initializing an *error accumulator* $\boldsymbol{z}(0) = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial x}(\hat{\boldsymbol{x}})$ and then accumulating the error by backpropagating $\boldsymbol{z}(j)$ through the $f_{\boldsymbol{w}}$ layer until its value converges to $\hat{\boldsymbol{z}}$. At each step j the additional error $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial x}(\hat{\boldsymbol{x}})$ is injected as was shown previously. If the state calculation converged, the error calculation is guaranteed to converge too. The number of iterations needed for the *error accumulator* $\boldsymbol{z}$ to converge can be different than the number of time steps needed for the state $\boldsymbol{x}$ to converge. In the conducted experiments it was usually much smaller.

5.6. Contraction map

In the previous paragraphs, it was stated that a fixed point $\hat{\boldsymbol{x}}$ of function $F_{\boldsymbol{w}}$ is sought. However, how can it be assumed that such a fixed point exists for the function $F_{\boldsymbol{w}}$? How to assure that it will be reached by iterating $\boldsymbol{x} = F_{\boldsymbol{w}}(\boldsymbol{x})$ using a random initial $\boldsymbol{x}$?

Actually all of the above can be assured by making $F_{\boldsymbol{w}}$ a *contraction map*. A *contraction map* (a *non-expansive* map) is a function $F_{\boldsymbol{w}}$ for which $d(F_{\boldsymbol{w}}(\boldsymbol{x}_1),\ F_{\boldsymbol{w}}(\boldsymbol{x}_2)) \leq d(\boldsymbol{x}_1,\ \boldsymbol{x}_2)$, where $d(\boldsymbol{x},\ \boldsymbol{y})$ is a distance function. For this implementation a distance function $d(\boldsymbol{x},\ \boldsymbol{y}) = max_i(|x_i - y_i|)$ was chosen, as it is independent of the *state* size and therefore, of the number of nodes in a given graph. The *Banach Fixed Point Theorem* states that a contraction map $F_{\boldsymbol{w}}(\boldsymbol{x})$ has the following properties:

— it has a single fixed point $\hat{\boldsymbol{x}}$

— it converges to $\hat{\boldsymbol{x}}$ from every starting point $\boldsymbol{x}(t_0)$

— the convergence to $\hat{\boldsymbol{x}}$ is exponentially fast [8].

How to assure that $F_{\boldsymbol{w}}$, a function composed of neural network instances, is actually a contraction map? The authors propose to impose a penalty function whenever the elements of the Jacobian $\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}$ suggest that $F_{\boldsymbol{w}}$ isn't a contraction map anymore.

Let $\boldsymbol{A} = \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x},\ \boldsymbol{l})$ be a block matrix of size $N \times N$ with blocks of size $s \times s$, where N is the number of nodes in the processed graph and $|\boldsymbol{x}_n| = s$ is the state size for a single node. A single block $\boldsymbol{A}_{n,u}$ measures the influence of the node u on node n if an edge from u to n exists or is zeroed otherwise. Let's denote by I_u^j the influence of node u on the jth element of state $\boldsymbol{x}_n$ (Eq. 5.11). The penalty $p_{\boldsymbol{w}}$ added to the network error $e_{\boldsymbol{w}}$ is defined by Eq. 5.9.

$$p_{\boldsymbol{w}} = \sum_{u \in N} \sum_{j=1}^{s} L(I_u^j,\ \mu) \tag{5.9}$$

$$L(y,\ \mu) = \begin{cases} y - \mu & \text{if } y > \mu \\ 0 & \text{otherwise} \end{cases} \tag{5.10}$$

$$I_u^j = \sum_{(n,u)} \sum_{i=1}^{s} |\boldsymbol{A}_{n,u}^{i,j}| \tag{5.11}$$

Basing on these equations, the value of $\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ is calculated and the final error derivative for the $h_{\boldsymbol{w}}$ network is calculated as $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} + \frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$. It can be seen from Eq. 5.10 that the term $\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ affects only weights that cause an excessive impact (larger than μ) and the value of such penalty is proportional to the value of I_u^j. The eagerness to impose the penalty and the penalty value are inversely proportional to the value of the *contraction constant* μ. The impact of *contraction constant* on the training process was described in section 6.3.

5.7. RPROP algorithm

The authors of the GNN algorithm suggest the RPROP algorithm [24] as the weights update algorithm, as an efficient gradient descent strategy. The basic idea of the RPROP algorithm is to use only the sign of the original weight updates $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$. The actual weight updates are calculated by the RPROP algorithm according to the past behaviour of $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$, which includes fast descent of monotonous gradient slopes, small steps in the proximity of a minimum and reverting updates that caused jumping over a local minimum. Actually, in the case of GNN, the RPROP algorithm should be used not only for its efficiency, but also as a way of dealing with the unpredictable behaviour of the $\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ term. As experiments showed, the value of $\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ can be larger than the original $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ by several orders of magnitude, which could disturb severely the learning algorithm if the RPROP algorithm wasn't used. The RPROP algorithm was implemented using standard recommended values [24], with the exception of Δ_{max} which was set to 1 to avoid large weight changes.

5.8. Maximum number of iterations

As was shown in section 5.6, the number of *Forward* or *Backward* steps is finite if $F_{\boldsymbol{w}}$ is a contraction map. However, it can be seen, that the penalty imposed on the weights is in fact imposed post factum. Only after the norm of the Jacobian $\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}$ increases excessively, the penalty is imposed. In fact:

1. it is not guaranteed that the Jacobian norm will be correct after the penalty is imposed
2. one *Forward* iteration takes place before the penalty is imposed

Even if the penalty is efficient enough (it isn't necessarily so, as shown in the subsequent experiments), the problem of a single *Forward* iteration that may not converge still remains. During experiments it was observed that while usually the number of *Forward* iterations was between 5 and 50, depending on the dataset, from time to time it reached about 2000 iterations or even more (the calculations had to be aborted due to excessive time). To make the calculation time predictable it was necessary to introduce a modification to the original GNN algorithm - a maximum number of *Forward* iterations. The value chosen for the subsequent experiments was 200, as it seemed to be a value large enough (in comparison to the usual number of steps) to assure that the state calculation will converge if $F_{\boldsymbol{w}}$ is still a contraction map. Furthermore, if the state calculation doesn't converge, it is not guaranteed that the error accumulation calculation will. Therefore, another similar restriction was introduced for the *Backward* procedure and the maximum number of *Backward* (error accumulation) iterations was set to 200.

5.9. Detailed training algorithm

```
1  MAIN:
       w = initialize
3      x = FORWARD(w)
4      for numberOfIterations:
5          [∂eh_w/∂w; ∂eg_w/∂w] = BACKWARD(x, w)
6          w = rprop-update(w, ∂eh_w/∂w, ∂eg_w/∂w)
7          x = FORWARD(w)
8      end
       return w
10 end

   FORWARD(w):
13     x(0) = random
14     t = 0
15     repeat:
           x(t+1) = F_w(x(t), l)
17         t = t+1
18     until (max_i(|x_i(t+1) − x_i(t)|) ≤ minStateDiff) or (t > maxForwardSteps)
19     return x(t)
20 end

   BACKWARD(x, w):
23     o = G_w(x)
24     A = ∂F_w/∂x (x, l)
25     b = ∂e_w/∂o · ∂G_w/∂x (x)
26     z(0) = b
27     t = 0
28     repeat:
           z(t−1) = z(t) · A + b
30         t = t−1
31     until (max_i(|z_i(t−1) − z_i(t)|) ≤ minErrorAccDiff) or (|t| > maxBackwardSteps)
32     ∂eg_w/∂w = b
33     ∂eh_w/∂w = z(t) · ∂F_w/∂w (x, l) + ∂p_w/∂w
34     return [∂eh_w/∂w; ∂eg_w/∂w]
35 end
```

Listing 5.1. The learning algorithm

5.10. Graph-focused tasks

The GNN model can be used for graph-focused tasks by modifying the standard learning algorithm. In a graph-focused task an output o_g is sought for every graph, which is the output of a predefined root node. In such a task only the root output error can be measured, as for every other node the expected output is not defined. Thus, the only modification necessary to deal with such a task is to set the error of all non-root nodes to zero. Such a modification was implemented and proved to work well for a modified subgraph matching task (see chapter 6), where the modified task consisted of determining if a given graph contains the expected subgraph S or not instead of selecting nodes belonging to S.

6. Experiments

Experiments were conducted to check if the implemented GNN is able to cope with the tasks presented in the original article [8]. For all the experiments the state size was set to 5, the number of hidden neurons in both the h_w and g_w networks was set to 5. After some successful trivial experiments, consisting of memorizing a single graph, the proper experiments were conducted. The task chosen for experiments was the *subgraph matching* task. It was chosen, because:

1. a similar experiment was conducted by Scarselli et al. [8]
2. the dataset is easy to generate, yet the problem is not trivial
3. to yield good results, the structure of the graph have to be exploited.

6.1. Subgraph matching - data

The datasets for the subgraph matching task were generated as follows. For a given number of graph nodes, graphs were generated by selecting node labels from $[0..10]$ and connecting each node pair in a graph with an edge probability δ. Then, edges were inserted randomly until the graph became connected. Then, a smaller (connected) subgraph S was inserted to every graph in the dataset. Then, a brute force algorithm was used to locate all copies of the subgraph S in every graph in the dataset. Thus, every graph in the dataset contained at least on copy of the subgraph S. Afterwards, a small Gaussian noise with zero mean and standard deviation of 0.25 was added to all node labels. All graph edges were undirected and thus were transformed to pairs of directed edges prior to processing. No edge labels were used.

Two datasets were generated. One with graph size (number of nodes) equal to 6, the subgraph size equal to 3 and $\delta = 0.8$ (100 graphs, called later the *6-3 dataset*). The second dataset with graph size equal to 14, subgraph size equal to 7 and $\delta = 0.2$ (100 graphs, called later the *14-7 dataset*). A larger δ was used for the first dataset, as graphs generated with $\delta = 0.2$ were mostly sequences. The first dataset was used to analyze the process of training, while the second one was used for comparison of GNN with a standard FNN classifier.

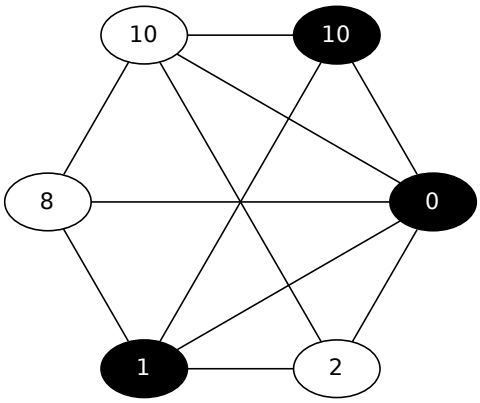

Figure 6.1. Sample graph from 6-3 dataset (subgraph in black), before adding noise

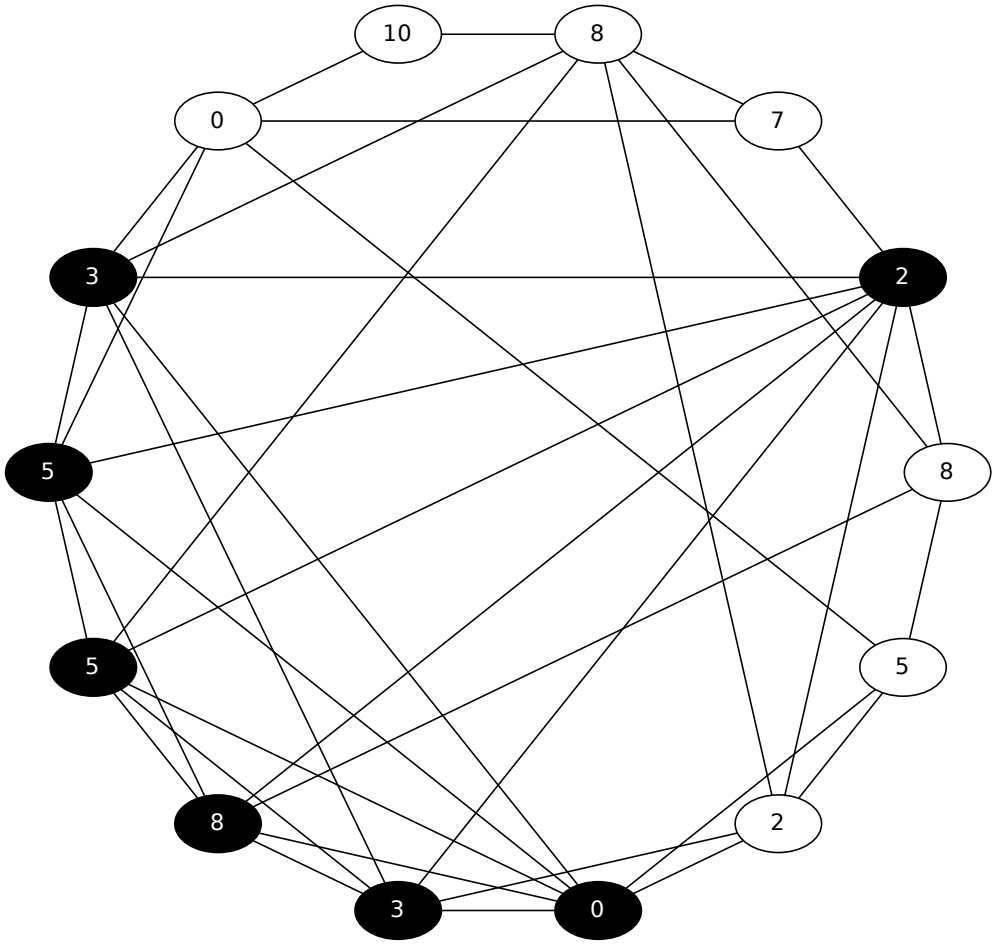

Figure 6.2. Sample graph from 14-7 dataset (subgraph in black), before adding noise

6.2. Impact of initial weight values on learning

To test the impact of initial weight values on the process of a GNN training, 9 different sets of weights were tested. For all tested networks, the contraction constant (μ from Eq. 5.9) was set to 0.9. The training was performed on 10 graphs belonging to the 6-3 dataset. Each GNN network was trained for 50 iterations. As the default error measure used in GNN training is the Mean Square Error, a similar performance measure - RMSE was used for evaluation. Results are presented in Fig. 6.3. Out of 9 networks, only 4 performed well: gnn2, gnn3, gnn5 and gnn7. The gnn5 network yielded the smallest RMSE at the end of training and also presents a remarkably monotonous RMSE slope compared to gnn7. All the other networks didn't improve significantly on the RMSE value, which may suggest that multiple initial sets of weights should be tried for a given dataset to build an efficient classifier.

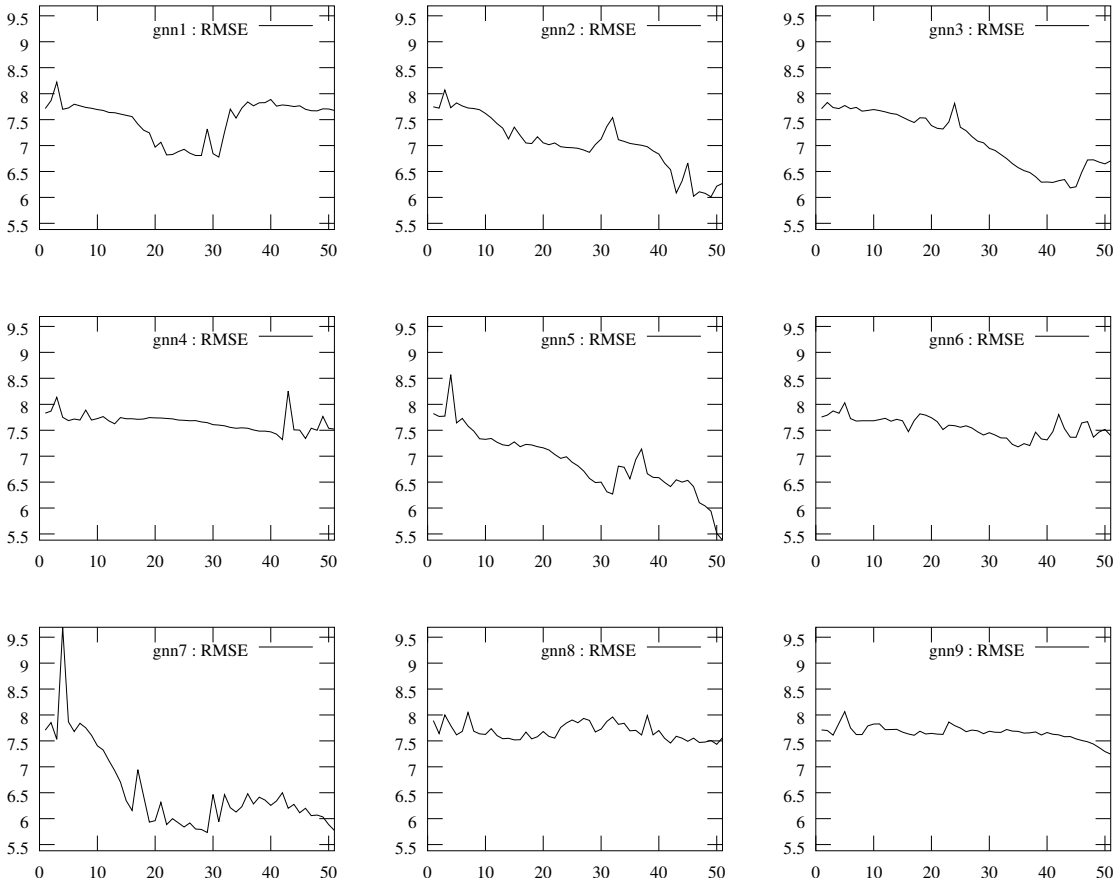

Figure 6.3. RMSE for 9 different initial weight sets. $\mu = 0.9$

6.3. Impact of contraction constant on learning

During the initial experiments, interesting results were obtained for different values of the contraction constant (μ from Eq. 5.9). It seems that for a given learning task exists a minimum value of μ below which no learning occurs. Some experiments were conducted for the 6-3 dataset using the best networks from Fig. 6.3: the gnn5 and gnn7 network (initial weight values were used). The results for gnn7 are presented in Fig. 6.4 and the results for gnn5 are presented in Fig. 6.5. For both networks three different values of μ were tested: 1.2, 0.9 and 0.6. In both cases it can be observed that no training occurs for $\mu = 0.6$. For these experiments 20 graphs from the 6-3 dataset were used.

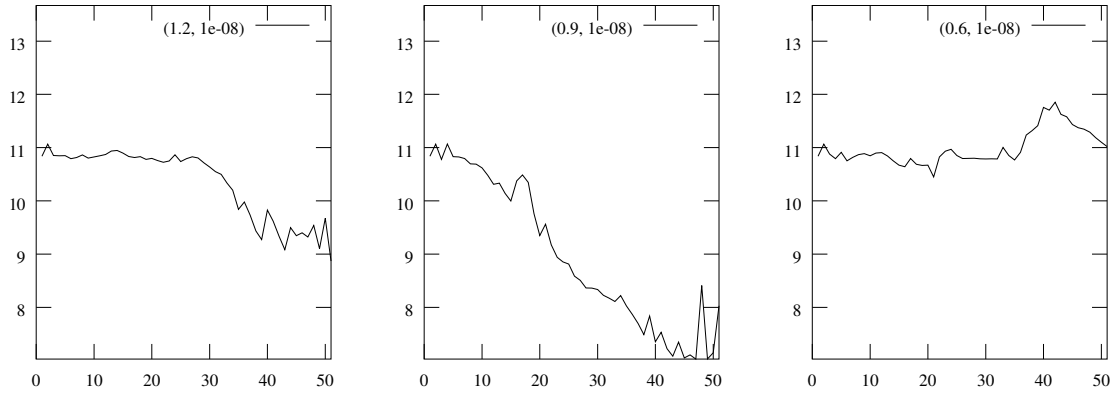

Figure 6.4. RMSE for gnn7 with $\mu \in [1.2, 0.9, 0.6]$

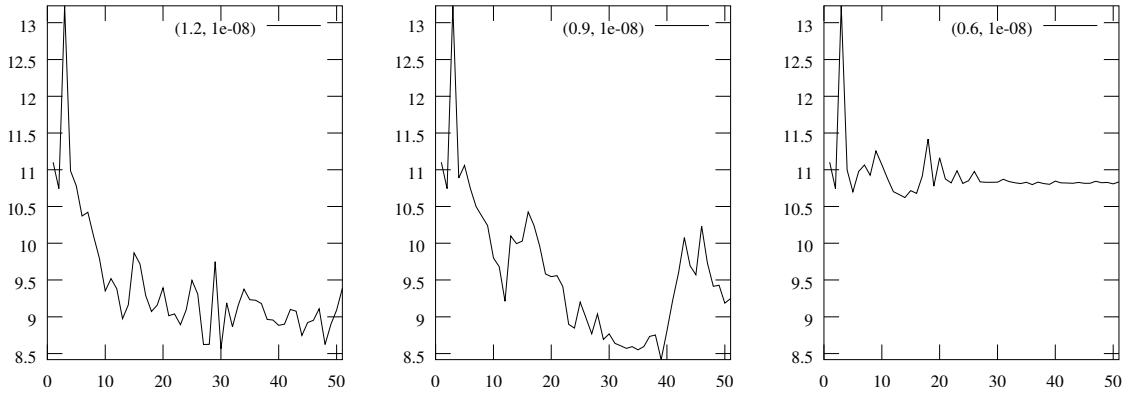

Figure 6.5. RMSE for gnn5 with $\mu \in [1.2, 0.9, 0.6]$

A closer look on the process of learning may shed some light on the reasons behind the lack of learning. In Fig. 6.6 the process of learning of gnn5 with $\mu = 0.9$ is presented. In Fig. 6.7 the same network gnn5 was trained with $\mu = 0.6$. The different values shown are: *nForward* - number of *Forward* (state building) iterations, *nBackward* - number of *Backward* (error accumulation) iterations, *penalty* - set to 1 if any weight was penalized, *de/dw influence* - percent of combined weight updates that had the same sign as $\frac{\partial e}{\partial w}$ (before passing to RPROP algorithm), *dp/dw influence* - percent of combined weight updates that had the same sign as

$\frac{\partial p}{\partial w}$. Some interesting features of the GNN model learning schema can be observed. In the case of $\mu = 0.9$ the number of *Forward* steps reached the maximum a couple of times, which presumably means that at that time the $F_{\boldsymbol{w}}$ ceased being a contraction map. The penalty was imposed mostly for short periods of time and only at one moment caused the $\frac{\partial e}{\partial w}$ influence to drop below 50%. This strategy yielded good results - the imposed penalty reduced the number of *Forward* steps and the RMSE was successfully reduced.

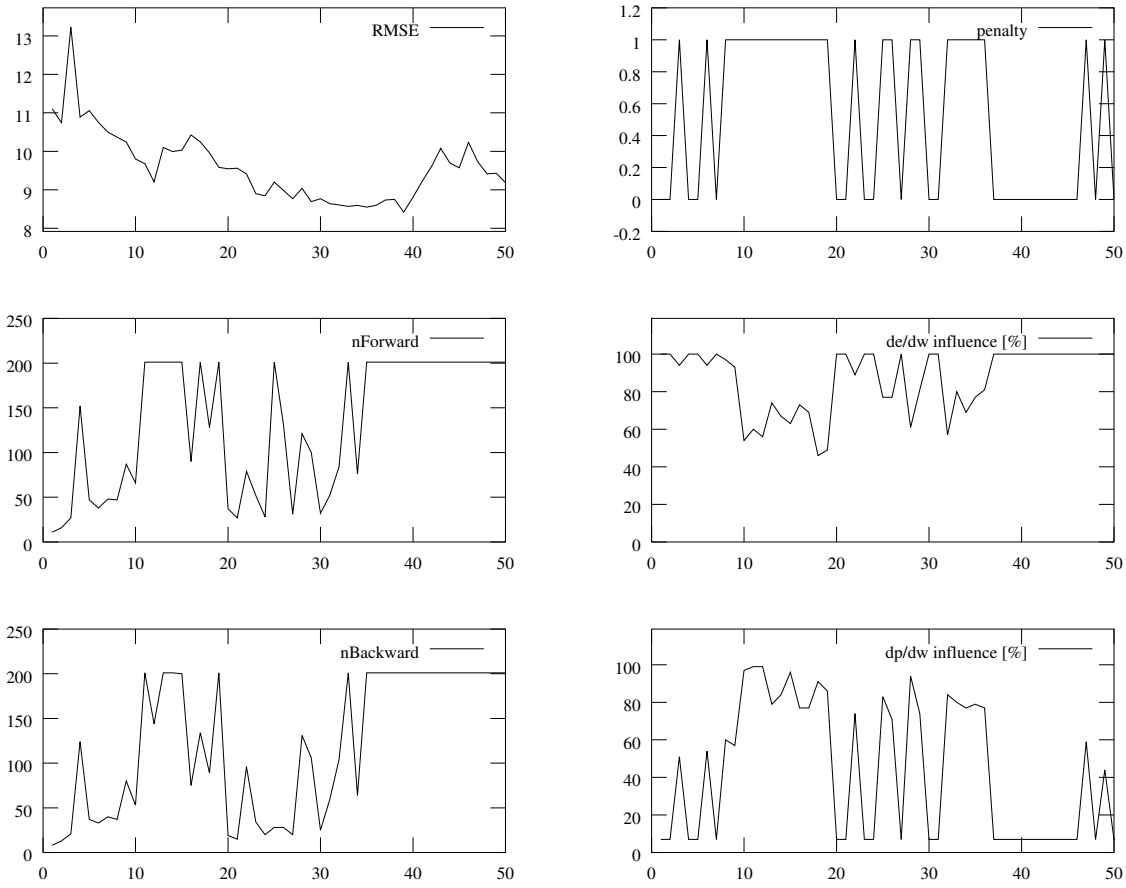

Figure 6.6. gnn5 performance with $\mu = 0.9$ for 20 graphs

A different situation is shown for $\mu = 0.6$. Because of a low μ value, the penalty was imposed eagerly and was larger than in the previous case (the impact of the μ value was described in section 5.6). It was imposed even when the number of *Forward* steps was below the maximum, that is when $F_{\boldsymbol{w}}$ was still a contraction map. Large values of the penalty caused a huge decrease of the $\frac{\partial e}{\partial w}$ term influence, which made any learning impossible.

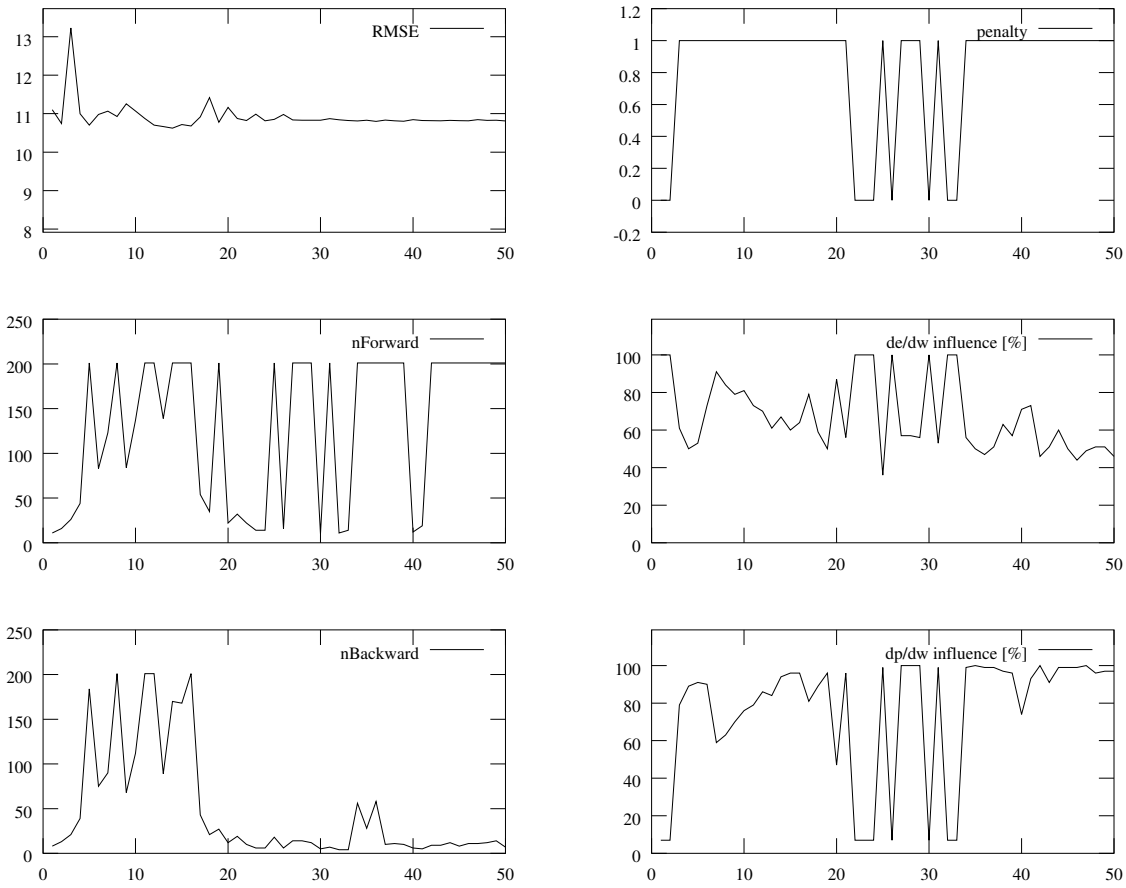

Figure 6.7. gnn5 performance with $\mu = 0.6$ for 20 graphs

Another interesting case is presented in Fig. 6.8: the learning process of gnn5 on 10 graphs with $\mu = 0.9$. It can be observed that even as the number of *Forward* steps reached in peaks the maximum value, the $F_{\boldsymbol{w}}$ function remained a contraction map. A large enough μ prevented the penalty from being imposed, which enabled the GNN model to train both computation units without any disturbance. The result is a monotonously decreasing RMSE slope, which could be previously observed in Fig. 6.3. It can be concluded, that the most important aspect of building a GNN model is to provide an efficient way to make $F_{\boldsymbol{w}}$ a contraction map as fast as possible, so as to provide as much time as possible for undisturbed learning.

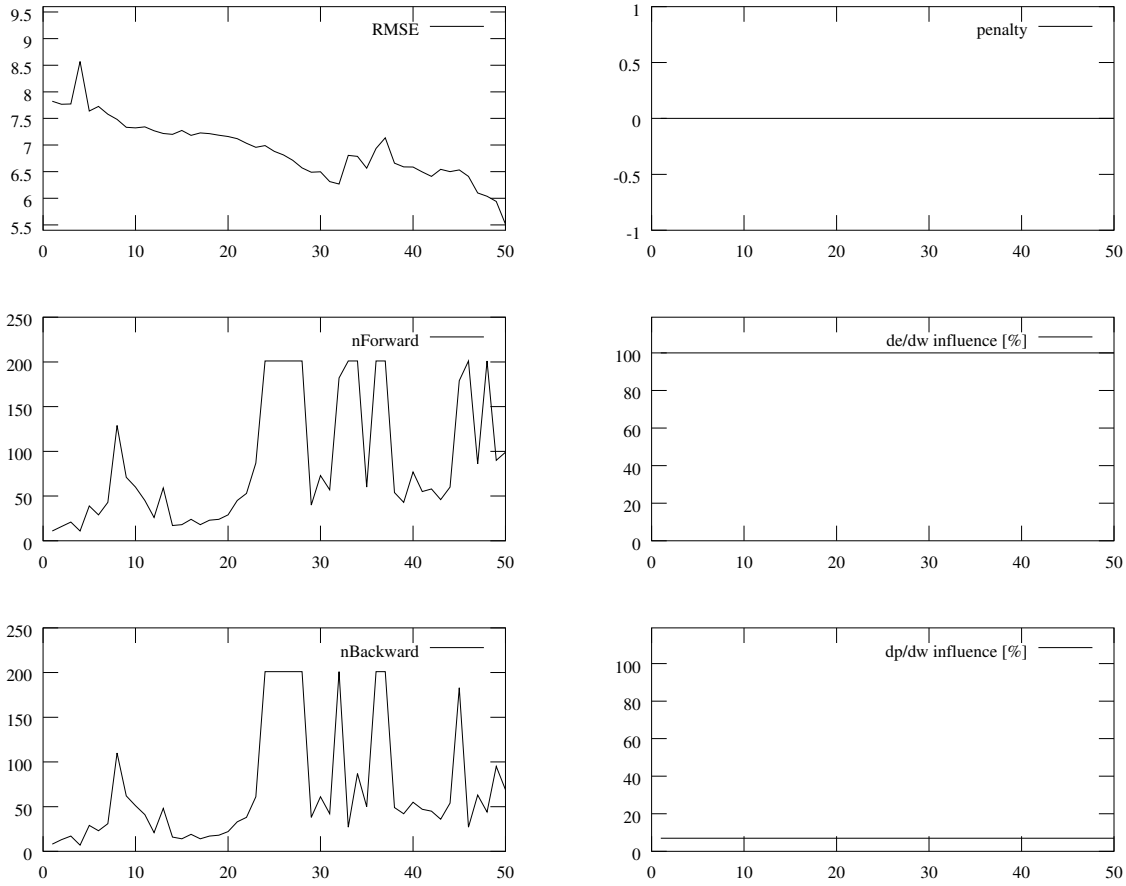

Figure 6.8. gnn5 performance with $\mu = 0.9$ for 10 graphs

6.4. Cross-validation results

To compare the performance of the implemented GNN model with a standard FNN, the following subgraph matching experiment was conducted. 5-fold cross-validation was performed on all 100 graphs from the 14-7 dataset. A random GNN was generated. It was trained with a contraction constant $\mu = 0.9$ for 50 iterations for each fold. To provide good FNN results, 10 three-layer FNNs with 20 hidden *tanh* neurons were evaluated and the one with best mean accuracy was selected. The results are presented in Table 6.1 and 6.2. The GNN classifier outperformed the FNN by more than 15%. This is due to the fact, that the FNN classifier could make predictions only by analyzing node labels, while the GNN classifier exploited correctly the graph topology.

These results can be better understood by analyzing the classified dataset. The 100 processed graphs consisted in total of 1400 nodes. Amongst these nodes, 1031 had node labels matching the subgraph node labels. Amongst these 1031 nodes only 702 actually belonged to the subgraph. Thus, 329 nodes, 23.5% of all the nodes would probably be classified as false positives by a classifier taking into consideration only node labels. This hypothesis corresponds quite well with the results presented.

	accuracy	precision	recall
FNN - tr	75%	68%	93%
FNN - tst	74%	68%	93%
GNN - tr	91%	87%	97%
GNN - tst	91%	86%	97%

Table 6.1. Mean values on training and test sets

	accuracy	precision	recall
FNN - tr	0.68%	0.82%	1.43%
FNN - tst	3.20%	2.89%	1.85%
GNN - tr	1.62%	1.71%	2.07%
GNN - tst	3.06%	3.70%	1.39%

Table 6.2. Standard deviations on training and test sets

7. Conclusions

The implementation of the GNN model yielded promising experimental results for structured data and proved that it can exploit properly both node labels and the graph structure. Some changes in the original algorithm, as the maximum number of *Forward* and *Backward* iterations were introduced to assure a more predictable computation time. Conditions under which the model works efficiently were described. An important parameter of the GNN model, determining the training efficiency was identified: the contraction constant μ. The most important conclusions are listed below:

— training yields best results when $F_{\boldsymbol{w}}$ remains a contraction map
— if $F_{\boldsymbol{w}}$ definitely ceases to be a contraction map, there are no training effects
— remaining near the contraction state can still yield good results
— a fixed maximum number of *Forward*/*Backward* iterations can still yield good results
— imposing an unnecessary penalty should be avoided
— too large penalties (a too small μ) should be avoided
— the minimum value of μ should be tuned to the processed dataset
— the minimum value of μ can be tuned using a subset of the data.

A. Using the software

The GNN classifier was implemented in GNU Octave 3.6.2 and tested on a x86_64 PC. The most important functions are:

— loadgraph - loads a single graph from .csv files
— loadset - loads a set of graphs sharing the same filename prefix
— mergegraphs - merge a cellarray of graphs to a single graph
— initgnn - initialize a new GNN
— traingnn - train GNN using a training graph
— classifygnn - classify given graph with a trained GNN
— evaluate - evaluate the classification results
— crossvalidate - perform cross-validation using an untrained GNN and a set of graphs

Help and usage information for each function can be displayed by typing *help <function_name>* in the Octave command line.

```
g6 = loadset('../data/g6s3n/g6s3', 10);
gm = mergegraphs(g6);
gnn = initgnn(gm.maxIndegree, [5 5], [5 gm.nodeOutputSize], 'tansig');
trainedGnn = traingnn(gnn, gm, 20);
outputs = classifygnn(trainedGnn, gm);
stats = evaluate(outputs, gm.expectedOutput);
```

Listing A.1. Sample usage session

All subgraph matching datasets were created using the *buildgraphs.py* script. Each graph can be viewed as a pdf file by using the *drawgraph.py* script. Each graph is stored as three .csv files, containing node labels, edge labels and expected outputs. A sample graph *'test'* is presented below. Nodes yielding output 2 were marked as black.

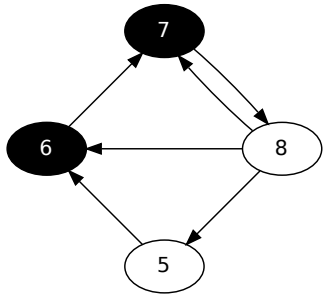

Figure A.1. Sample graph

test_nodes.csv	test_output.csv	test_edges.csv
5	1	1,2,0
6	2	2,3,0
7	2	3,4,0
8	1	4,1,0
		4,2,0
		4,3,0

Table A.1. Sample graph data

B. Listings

```
1
function [bestGnn trainStats] = traingnn(gnn, graph, nIterations, ...
  maxForwardSteps=200, maxBackwardSteps=200, initialState=0)
% Trains GNN using graph as training set
5 %
% usage: [bestGnn trainStats] = traingnn(gnn, graph, nIterations,
% maxForwardSteps=200, maxBackwardSteps=200, initialState=0)
%
% return:
10 % - best gnn obtained during training and all errors
%

  if initialState != 0
    assert(size(initialState, 1) == graph.nNodes);
15   assert(size(initialState, 2) == gnn.stateSize);
  end

  % constants for indexing training stats
  RMSE = 1;
20 FORWARD_STEPS = 2;
  BACKWARD_STEPS = 3;
  PENALTY_ADDED = 4;
  RPROP_REVERTED_TRANS = 5;
  RPROP_REVERTED_OUT = 6;
25 TRANS_STATS_START = 7;
  TRANS_STATS_END = TRANS_STATS_START + 3;
  trainStats = zeros(nIterations + 1, TRANS_STATS_END);

  % normalize edge and node labels
30 % store normalization info inside result gnn
  [graph.nodeLabels gnn.nodeLabelMeans gnn.nodeLabelStds] = ...
    normalize(graph.nodeLabels);
  [graph.edgeLabels(:, 3:end) gnn.edgeLabelMeans gnn.edgeLabelStds] = ...
    normalize(graph.edgeLabels(:, 3:end));
35 graph = addgraphinfo(graph);

  minError = Inf;
  bestGnn = gnn;
  rpropTransitionState = initrprop(gnn.transitionNet);
40 rpropOutputState = initrprop(gnn.outputNet);
  for iteration = 1:nIterations
    [state nForwardSteps] = forward(gnn, graph, maxForwardSteps, ...
      initialState);
    trainStats(iteration, FORWARD_STEPS) = nForwardSteps;
45
    outputs = applynet(gnn.outputNet, state);
    if graph.nodeOrientedTask == false
      err = rmse(graph.expectedOutput(1, :), outputs(1, :));
    else
```

```
50        err = rmse(graph.expectedOutput, outputs);
        end
        trainStats(iteration, RMSE) = err;
        if err < minError
          minError = err;
55        bestGnn = gnn;
        end

        [deltas nBackwardSteps penaltyAdded] = backward(gnn, graph, ...
          state, maxBackwardSteps);
60      trainStats(iteration, BACKWARD_STEPS) = nBackwardSteps;
        trainStats(iteration, PENALTY_ADDED) = penaltyAdded;
        trainStats(iteration, TRANS_STATS_START:TRANS_STATS_END) = ...
          round(transitionstats(deltas));

65      outputDerivatives = deltas.output;
        [rpropOutputState outputWeightUpdates nOutputReverted] = ...
          rprop(rpropOutputState, outputDerivatives);
        trainStats(iteration, RPROP_REVERTED_OUT) = ...
          round(nOutputReverted * 100 / gnn.outputNet.nWeights);
70
        transitionDerivatives = adddeltas(deltas.transition, ...
          deltas.transitionPenalty);
        [rpropTransitionState transitionWeightUpdates ...
          nTransitionReverted] = ...
75          rprop(rpropTransitionState, transitionDerivatives);
        trainStats(iteration, RPROP_REVERTED_TRANS) = ...
          round(nTransitionReverted * 100 / gnn.transitionNet.nWeights);

        gnn.outputNet = updateweights(gnn.outputNet,...
80        outputWeightUpdates, 1);
        gnn.transitionNet = updateweights(gnn.transitionNet,...
          transitionWeightUpdates, 1);
      end

85    % calculate RMSE of final GNN
      iteration = nIterations + 1;

      [state nForwardSteps] = forward(gnn, graph, maxForwardSteps, ...
        initialState);
90    trainStats(iteration, FORWARD_STEPS) = nForwardSteps;

      outputs = applynet(gnn.outputNet, state);
      if graph.nodeOrientedTask == false
        err = rmse(graph.expectedOutput(1, :), outputs(1, :));
95    else
        err = rmse(graph.expectedOutput, outputs);
      end
      trainStats(iteration, RMSE) = err;
      %printf('RMSE after %d iterations: %f\n', iteration - 1, err);
100   if err < minError
        minError = err;
        bestGnn = gnn;
      end
    end
```

Listing B.1. Main train function

```
function [state nSteps] = forward(gnn, graph, maxForwardSteps, state=0)
% Perform the 'forward' step of GNN training
% compute node states until stable state is reached
%
% usage: [state nSteps] = forward(gnn, graph, maxForwardSteps, state=0)

  if state == 0
    state = initstate(graph.nNodes, gnn.stateSize);
  end
  nSteps = 0;
  do
    if nSteps > maxForwardSteps
      % printf('Too many forward steps: %d, aborting\n', nSteps);
      return;
    end
    lastState = state;
    state = transition(gnn.transitionNet, lastState, graph);
    nSteps = nSteps + 1;
  until(stablestate(lastState, state, gnn.minStateDiff));
end
```

Listing B.2. Forward function

```
function newState = transition(fnn, state, graph)
% Calculate global transition function

  newState = zeros(size(state));
  for nodeIndex = 1:graph.nNodes
    % build transitionNet input for single node
    sourceNodeIndexes = graph.sourceNodes{nodeIndex};
    nodeLabel = graph.nodeLabels(nodeIndex, :);
    newNodeState = zeros(1, fnn.nOutputNeurons);
    for i = 1:size(sourceNodeIndexes, 2)
      sourceEdgeLabel = ...
        graph.edgeLabelsCell{sourceNodeIndexes(i), nodeIndex};
      sourceNodeState = state(sourceNodeIndexes(i), :);
      inputs = [nodeLabel, sourceEdgeLabel, sourceNodeState];
      stateContribution = applynet(fnn, inputs);
      newNodeState = newNodeState + stateContribution;
    end
    % if node has < maxIndegree input nodes, its contribution is 0
    newState(nodeIndex, :) = newNodeState;
  end
end
```

Listing B.3. Transition function

```
function stable = stablestate(lastState, state, minStateDiff)
% return true, if difference between two states is insignificant
  diff = max(max(abs(lastState - state)));
  stable = (diff < minStateDiff);
end
```

Listing B.4. State convergence check

```
1
function [weightDeltas nSteps penaltyAdded] = backward(gnn, graph, state,
maxBackwardSteps)
% Perform the 'backward' step of GNN training
5 %
% usage: [weightDeltas nSteps penaltyAdded] = backward(gnn, graph, state,
% maxBackwardSteps)
%
% state : stable state, calculated by forward
10 % return : deltas for both transition and output networks of gnn

  outputs = applynet(gnn.outputNet, state);
  outputErrors = 2 .* (graph.expectedOutput - outputs);
  if graph.nodeOrientedTask == false
15   selectedNodeErrors = outputErrors(1, :);
    outputErrors = zeros(size(outputErrors));
    outputErrors(1, :) = selectedNodeErrors;
  end

20  outputDeltas = outputdeltas(gnn.outputNet, graph, state, outputErrors);

  % sparse matrix calculations, cause size(A) = N x N x s^2
  % A can contain whole training set as a single graph
  A = calculatea(gnn.transitionNet, graph, state);
25  b = sparse(calculateb(gnn.outputNet, graph, state, outputErrors));
  accumulator = b;  % accumulator contains dew/do * dG/wx
  nSteps = 0;
  do
    % if there are too many backward steps, we can get infinities
30    if nSteps > maxBackwardSteps
      break;
    end
    lastAccumulator = accumulator;
    % for each source node, backpropagate error of its target nodes
35    % A : influence of source nodes on target nodes
    % accumulator : errors de/dx from previous timestep (t + 1)
    % b : de/do * dG/dx error, injected at each step
    accumulator = accumulator * A + b;
    nSteps = nSteps + 1;
40  until(stablestate(lastAccumulator, accumulator, gnn.minErrorAccDiff));

  transitionErrors = reshape(accumulator, graph.nNodes, gnn.stateSize);
  transitionDeltas = transitiondeltas(gnn.transitionNet, graph, state,
transitionErrors);
45
  % calculate penalty dp/dw, assuring that F is a contraction map
  [penaltyDerivative penaltyAdded] = penaltyderivative(gnn,graph,state,A);
  penaltyDeltas = reshapedeltas(gnn.transitionNet, penaltyDerivative);

50  weightDeltas = struct(...
    'output', outputDeltas,...
    'transition', transitionDeltas,...
    'transitionPenalty', penaltyDeltas);
end
```

Listing B.5. Backward function

```
1
function A = calculatea(transitionNet, graph, state)
% Calculate the A(x) = dF/dx block matrix
% (NxN blocks, each sxs) (F = transition function)
5 %
% usage: A = calculatea(transitionNet, graph, state)
%
% Each block A[n,u] = dxn/dxu:
% - describes the effect of node xu on node xn,
10 % if an edge xu->xn exists
% - is null (zeroed) if there is no edge
%
% Each element of block A[n, u] : a[i, j]:
% - describes the effect of ith element of state xu
15 % on jth element of state xn

  stateSize = transitionNet.nOutputNeurons;
  aSize = graph.nNodes * stateSize;
  A = sparse(aSize, aSize); % zeroed
20  for nodeIndex = 1:graph.nNodes
    % build transitionNet input for single node
    sourceNodeIndexes = graph.sourceNodes{nodeIndex};
    nodeLabel = graph.nodeLabels(nodeIndex, :);
    for i = 1:size(sourceNodeIndexes, 2)
25      sourceEdgeLabel = ...
          graph.edgeLabelsCell{sourceNodeIndexes(i), nodeIndex};
        sourceNodeState = state(sourceNodeIndexes(i), :);
        inputs = [nodeLabel, sourceEdgeLabel, sourceNodeState];
        deltaZx = zeros(stateSize, stateSize);
30      for j = 1:stateSize
          errors = zeros(1, stateSize);
          errors(j) = 1;
          inputDeltas = bp2(transitionNet, inputs, errors);
          % select only weights corresponding to x_iu
35        stateWeightsStart = ...
            1 + graph.nodeLabelSize + graph.edgeLabelSize;
          stateInputDeltas = inputDeltas(1, stateWeightsStart:end);
          deltaZx(:, j) = stateInputDeltas';
        end
40      startX = blockstart(nodeIndex, stateSize);
        endX = blockend(nodeIndex, stateSize);
        startY = blockstart(sourceNodeIndexes(i), stateSize);
        endY = blockend(sourceNodeIndexes(i), stateSize);
        A(startX:endX, startY:endY) = deltaZx;
45    end
  end
end
```

Listing B.6. Matrix A calculation

```
1
function b = calculateb(outputNet, graph, state, errorDerivative)
% Calculate b matrix = dew/do * dGw/dx
%
5 % G : global output function, G(nodeStates)=outputs
% errorDerivative : each row contains an error of a node
% state : each row contains a node state

  b = zeros(outputNet.nInputLines, graph.nNodes);
10  for nodeIndex = 1:graph.nNodes
    errors = errorDerivative(nodeIndex, :);
    nodeState = state(nodeIndex, :);
    inputs = nodeState;
    deltas = backpropagate(outputNet, inputs, errors);
15    b(:, nodeIndex) = deltas.deltaInputs(1, :);
  end
  % stack output for each node on one another
  b = vec(b)';
end
```

Listing B.7. Matrix b calculation

```
1
function [penaltyDerivative penaltyAdded] =
  penaltyderivative(gnn, graph, state, A)
% Calculate penalty derivative contribution to de/dw, where:
% - e = RMSE + contractionMapPenalty (network error)
%
% usage: [penaltyDerivative penaltyAdded]
% = penaltyderivative(gnn, graph, state, A)
%

% sum up influence of each source node on all the other nodes
% each s-long block is influence of single source node xu on all s outputs
  sourceInfluences = sum(abs(A), 1);
  sourceInfluences = (sourceInfluences - ...
    repmat(gnn.contractionConstant, size(sourceInfluences))) .*...
    (sourceInfluences > gnn.contractionConstant);

  fnn = gnn.transitionNet;
  nWeights1 = fnn.nInputLines * fnn.nHiddenNeurons;
  nBias1 = fnn.nHiddenNeurons;
  nWeights2 = fnn.nOutputNeurons * fnn.nHiddenNeurons;
  nBias2 = fnn.nOutputNeurons;
  penaltyDerivative = zeros(1, nWeights1 + nBias1 + nWeights2 + nBias2);
  if sum(sourceInfluences) == 0
    penaltyAdded = false;
  else
    penaltyAdded = true;
    % matrix B contains influences from A, filtered:
    % only influences coming from a too influential source are retained
    B = sign(A) .* repmat(sourceInfluences, size(A, 1), 1);
    for sourceIndex = 1:graph.nNodes
      startX = blockstart(sourceIndex, gnn.stateSize);
      endX = blockend(sourceIndex, gnn.stateSize);
      sourceNodeInfluences = sourceInfluences(1, startX:endX);
      if (sum(sourceNodeInfluences, 2) != 0)
        % calculate impactDerivative[n, u] for u = sourceIndex
        for targetIndex = 1:graph.nNodes
          if !edgeexists(graph, sourceIndex, targetIndex)
            continue;
          end
          startY = blockstart(targetIndex, gnn.stateSize);
          endY = blockend(targetIndex, gnn.stateSize);
          Rnu =  full(B(startY:endY, startX:endX));

          % calculate f2'(net2) and f1'(net1)
          nodeLabel = graph.nodeLabels(targetIndex, :);
          sourceEdgeLabel = graph.edgeLabelsCell{sourceIndex, targetIndex};
          sourceNodeState = state(sourceIndex, :);
          inputs = [nodeLabel, sourceEdgeLabel, sourceNodeState];

          % fnn feed
          net1 = fnn.weights1 * inputs' + fnn.bias1;
          hiddenOutputs = fnn.activation1(net1);
          net2 = fnn.weights2 * hiddenOutputs + fnn.bias2;
          sigma1 = fnn.activationderivative1(net1);
          sigma2 = fnn.activationderivative2(net2);

          % select only weights corresponding to xu at inputs
```

```
            stateWeightsStart = 1 + graph.nodeLabelSize + graph.edgeLabelSize;
            stateWeights1 = fnn.weights1(:, stateWeightsStart:end);

            % vec(Rnu)' * dvec(Anu)/dw = da1 + da2 + da3 + da4
            deltaSignal1 = vec(Rnu * stateWeights1' * ...
              diag(sigma1) * fnn.weights2')' * vecdiagmatrix(gnn.stateSize);
            fnn2nd = fnn;
            fnn2nd.activation1 = fnn.activationderivative1;
            fnn2nd.activation2 = fnn.activationderivative2;
            fnn2nd.activationderivative1 = fnn.activation2ndderivative1;
            fnn2nd.activationderivative2 = fnn.activation2ndderivative2;
            deltas1 = backpropagate(fnn2nd, inputs, deltaSignal1);
            da1 = vecdeltas(deltas1)';

            da2left = vec(diag(sigma2) * Rnu * stateWeights1' * diag(sigma1))';
            weights2dw = [zeros(nWeights2, nWeights1 + nBias1) ...
              eye(nWeights2) zeros(nWeights2, nBias2)];
            da2 = da2left * weights2dw;

            deltaSignal3 = vec(fnn.weights2' * diag(sigma2) * ...
              Rnu * stateWeights1')' * vecdiagmatrix(fnn.nHiddenNeurons);
            % 1-layer fnn backpropagate with f(net) = transition.f'(net)
            net1 = fnn.weights1 * inputs' + fnn.bias1;
            delta1 = deltaSignal3' .* fnn.activation2ndderivative1(net1);
            deltaWeights1 = delta1 * inputs;
            deltaBias1 = delta1;
            da3 = [vec(deltaWeights1); vec(deltaBias1); ...
              zeros(nWeights2 + nBias2, 1)]';

            da4left = vec(diag(sigma1) * fnn.weights2' * diag(sigma2) * Rnu)';
            stateSize = fnn.nOutputNeurons;
            nStateWeights = stateSize * fnn.nHiddenNeurons;
            labelsSize = graph.nodeLabelSize + graph.edgeLabelSize;
            stateWeights1Dw = zeros(nStateWeights, ...
              nWeights1 + nBias1 + nWeights2 + nBias2);
            % mark with ones weights corresponding to xu
            for h = 1:fnn.nHiddenNeurons
              startIndexX = 1 + (h - 1) * fnn.nInputLines + labelsSize;
              endIndexX = startIndexX + stateSize - 1;
              startIndexY = 1 + (h - 1) * stateSize;
              endIndexY = startIndexY + stateSize - 1;
              stateWeights1Dw(startIndexY:endIndexY, ...
                startIndexX:endIndexX) = eye(stateSize);
            end
            da4 = da4left * stateWeights1Dw;

            impactDerivative = da1 + da2 + da3 + da4;
            penaltyDerivative = penaltyDerivative + impactDerivative;
          end
        end
      end
      penaltyDerivative = penaltyDerivative * 2;
    end
  end
```

Listing B.8. Penalty derivative calculation

```
1
function [rpropStruct nReverted] =
  rpropupdate(rpropStruct, errorDerivatives, deltaMin, deltaMax)
% Helper function for rprop(), operates on sigle matrix of weights
5 %
% usage: [rpropStruct nReverted] =
% rpropupdate(rpropStruct, errorDerivatives, deltaMin, deltaMax)
%
% rpropStruct.weightUpdates - deltas that should be added to fnn weights
10


  assert(size(rpropStruct.errors, 1) == size(errorDerivatives, 1));
  assert(size(rpropStruct.errors, 2) == size(errorDerivatives, 2));
  increase = 1.2;
15 decrease = 0.5;
  errorDirectionChange = rpropStruct.errors .* errorDerivatives;

  nReverted = 0;
  for i = 1:size(rpropStruct.deltas, 1)
20   for j = 1:size(rpropStruct.deltas, 2)
      if errorDirectionChange(i, j) > 0
        % increase weight update
        rpropStruct.deltas(i, j) = min(rpropStruct.deltas(i, j) * ...
          increase, deltaMax);
25      rpropStruct.weightUpdates(i, j) = ...
          sign(errorDerivatives(i, j)) * rpropStruct.deltas(i, j);
        rpropStruct.errors(i, j) = errorDirectionChange(i, j);
      else if errorDirectionChange(i, j) < 0
        rpropStruct.deltas(i, j) = ...
30        max(rpropStruct.deltas(i, j) * decrease, deltaMin);
        % revert last weight update
        rpropStruct.weightUpdates(i, j) = ...
          -rpropStruct.weightUpdates(i, j);
        % avoid double punishment in next step
35      rpropStruct.errors(i, j) = 0;
        nReverted = nReverted + 1;
      else
        rpropStruct.weightUpdates(i, j) = ...
          sign(errorDerivatives(i, j)) * rpropStruct.deltas(i, j);
40      rpropStruct.errors(i, j) = errorDerivatives(i, j);
      end
    end
  end
end
```

Listing B.9. RPROP weights update

List of Figures

Listings

Bibliography

[1] J. B. Pollack, "Recursive distributed representations," *Artificial Intelligence*, vol. 46, no. 1, pp. 77–105, 1990.

[2] A. Sperduti, "Labelling recursive auto-associative memory," *Connection Science*, vol. 6, no. 4, pp. 429–459, 1994.

[3] A. Goulon-Sigwalt-Abram, A. Duprat, and G. Dreyfus, "From hopfield nets to recursive networks to graph machines: numerical machine learning for structured data," *Theoretical computer science*, vol. 344, no. 2, pp. 298–334, 2005.

[4] M. Bianchini, M. Gori, L. Sarti, and F. Scarselli, "Backpropagation through cyclic structures," in *AI* IA 2003: Advances in Artificial Intelligence*, pp. 118–129, Springer, 2003.

[5] O. Ivanciuc, "Canonical numbering and constitutional symmetry," *Handbook of Chemoinformatics: From Data to Knowledge in 4 Volumes*, pp. 139–160, 2003.

[6] M. Bianchini, M. Maggini, L. Sarti, and F. Scarselli, "Recursive neural networks for processing graphs with labelled edges: Theory and applications," *Neural Networks*, vol. 18, no. 8, pp. 1040–1050, 2005.

[7] P. Frasconi, M. Gori, and A. Sperduti, "A general framework for adaptive processing of data structures," *Neural Networks, IEEE Transactions on*, vol. 9, no. 5, pp. 768–786, 1998.

[8] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *Neural Networks, IEEE Transactions on*, vol. 20, no. 1, pp. 61–80, 2009.

[9] A. Goulon, T. Picot, A. Duprat, and G. Dreyfus, "Predicting activities without computing descriptors: graph machines for QSAR," *SAR and QSAR in Environmental Research*, vol. 18, no. 1-2, pp. 141–153, 2007.

[10] A. Goulon, A. Duprat, and G. Dreyfus, "Learning numbers from graphs," *Applied Statistical Modelling and Data Analysis, Brest, France*, pp. 17–20, 2005.

[11] S. Yong, M. Hagenbuchner, A. Tsoi, F. Scarselli, and M. Gori, "XML document mining using graph neural network," *Center for Computer Science http://inex. is. informatik. uni-duisburg. de/2006*, p. 354, 2006.

[12] F. Scarselli, S. L. Yong, M. Gori, M. Hagenbuchner, A. C. Tsoi, and M. Maggini, "Graph neural networks for ranking web pages," in *Web Intelligence, 2005. Proceedings. The 2005 IEEE/WIC/ACM International Conference on*, pp. 666–672, IEEE, 2005.

[13] G. Monfardini, V. Di Massa, F. Scarselli, and M. Gori, "Graph neural networks for object localization," *Frontiers in Artificial Intelligence and Applications*, vol. 141, p. 665, 2006.

[14] A. Quek, Z. Wang, J. Zhang, and D. Feng, "Structural image classification with graph neural networks," in *Digital Image Computing Techniques and Applications (DICTA), 2011 International Conference on*, pp. 416–421, IEEE, 2011.

[15] F. Costa, P. Frasconi, V. Lombardo, and G. Soda, "Towards incremental parsing of natural language using recursive neural networks," *Applied Intelligence*, vol. 19, no. 1-2, pp. 9–25, 2003.

[16] S. Saha and G. Raghava, "Prediction of continuous b-cell epitopes in an antigen using recurrent neural network," *PROTEINS: Structure, Function, and Bioinformatics*, vol. 65, no. 1, pp. 40–48, 2006.

[17] R. Kree and A. Zippelius, "Recognition of topological features of graphs and images in neural networks," *Journal of Physics A: Mathematical and General*, vol. 21, no. 16, p. L813, 1988.

[18] A. Küchler and C. Goller, "Inductive learning in symbolic domains using structure-driven recurrent neural networks," in *KI-96: Advances in Artificial Intelligence*, pp. 183–197, Springer, 1996.

[19] A. Stolcke and D. Wu, *Tree matching with recursive distributed representations*. Citeseer, 1992.

[20] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Neural Networks, 1996., IEEE International Conference on*, vol. 1, pp. 347–352, IEEE, 1996.

[21] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Physical review letters*, vol. 59, no. 19, pp. 2229–2232, 1987.

[22] R. J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," *Back-propagation: Theory, architectures and applications*, pp. 433–486, 1995.

[23] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *Neural Networks, IEEE Transactions on*, vol. 8, no. 3, pp. 714–735, 1997.

[24] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in *Neural Networks, 1993., IEEE International Conference on*, pp. 586–591, IEEE, 1993.