Recurrent Neural Collective Classification

Derek D. Monner, James A. Reggia

Abstract—With the recent surge in availability of datasets containing not only individual attributes but also relationships, classification techniques that take advantage of predictive relationship information have gained in popularity. The most popular existing collective classification techniques have a number of limitations—some of them generate arbitrary and potentially lossy summaries of the relationship data, while others ignore directionality and strength of relationships. Popular existing techniques make use of only direct neighbor relationships when classifying a given entity, ignoring potentially useful information contained in expanded neighborhoods of radius greater than one. We present a new technique that we call recurrent neural collective classification (RNCC) which avoids arbitrary summarization, utilizes information about relationship directionality and strength, and, through recursive encoding, learns to leverage larger relational neighborhoods around each entity. Experiments with synthetic datasets show that RNCC can make effective use of relationship data for both direct and expanded neighborhoods. Further experiments demonstrate that our technique outperforms previously published results of several collective classification methods on a number of real-world datasets.

Index Terms—collective classification, expanded relational neighborhoods, recurrent neural network, long short-term memory architecture

I. INTRODUCTION

In recent years there has been a growing interest in classification techniques that can take advantage of relationships between the entities being classified. In particular, consider cases where we are concerned with a large set of similar entities, and we know the relationships these entities have to one another. Systems that utilize these relationships in determining class assignments have been dubbed collective classification techniques. Such techniques are useful, for example, for classifying academic papers based on their citations to the wider literature, or for determining which people are likely to be influential in a social network based on who they communicate with. However, the existing techniques for collective classification have a number of limitations, including information loss through summarization, blindness to relationship strength and directionality, and myopia with respect to indirect relationships between entities—manifested as chains of links in the relationship graph.

In a traditional classification problem, one must map an entity to one of a few pre-defined classes. In collective classification, the set of entities to be classified is accompanied by a graph specifying relationships between the entities. The relationships in this graph augment each entity’s individual features with additional information, generally gleaned from the entity’s direct neighbors in the graph—those that share a directed or undirected edge of some type with the entity. This additional information takes varying forms for different techniques. The iterated classification algorithm (ICA) [1] aggregates neighbor features and/or classes using a summarizing function like sum or mode to extend each entity’s feature set with neighbor data; this new set of features can be run through traditional classifiers. When this supplementary information includes the predicted class of each neighbor, the extended feature set for each entity can be updated iteratively as the classifier improves in performance during training. A similar approach is to use Gibbs sampling (GS) [2, 3] combined with the output of a local classifier to similarly update an expanded set of features describing the neighbors. Other approaches focus on statistical models that directly represent probabilities of a given class co-occurring with both the individual features and the relationships. For example, both loopy belief propagation (LBP) [4, 5] and mean-field relaxation labeling (MFRL) [6] perform approximate inference on an underlying pairwise Markov random field (MRF) [7] to guess classes based on the features of the entity and those of its direct neighbors. However, since this underlying model supports only undirected graphs, these approaches are incapable of taking into account potentially important information about relationship directionality.

Furthermore, the techniques mentioned above use only direct neighbor relationships to facilitate predictions. For some applications, expanded relationship neighborhoods of radius greater than one around each entity may provide critical information that these techniques could miss. For example, if our goal is to determine which of a group of fiction authors will become successful, it may matter less who an author’s agent is than to whom the agent herself is connected.

This paper presents a new technique called recurrent neural collective classification (RNCC) that aims to avoid the limitations presented above. RNCC eschews the summarization used by ICA and GS in favor of sequential presentation of full neighbor feature-sets. Unlike LBP and MFRL, RNCC is capable of taking advantage of directionality and strength of relationships. Finally, the neural network used in RNCC develops a learned representation for each entity to be classified which incorporates information about its neighbors. This is important because these representations can be fed to the network as additional neighbor inputs, allowing our technique to take advantage of relevant information gleaned from expanded relational neighborhoods without incurring the computational expense of directly examining those neighborhoods. These features of RNCC allow it to achieve better performance than existing techniques on real-world datasets.

In the next section we describe existing collective classification approaches in more detail, contrasting them with RNCC, which is then described fully in Section II. In Section IV we
describe a number of experimental results comparing RNCC to the best published results on common real-world datasets before proceeding to a brief discussion in Section III.

II. EXISTING COLLECTIVE CLASSIFICATION TECHNIQUES

In a collective classification problem, we are given a set of entities to classify. Each entity is associated with a fixed-length vector of entity features which describe the characteristics of the entity. For example, if the entities are documents, the entity features for a given document might include normalized term frequency counts for each word it contains. Each of these features may be binary or real-valued, where the real values are often scaled to the range $[0, 1]$; multinomial features with $n$ possible values can be transformed into $n$ binary features, of which one is active at a time. Each entity is also associated with one or more classes that our technique needs to predict. Continuing with the documents-as-entities example, an entity’s class might be its topic, its author, the name of the journal in which it appeared, or any other missing bit of information.

So far we could be describing most any classification problem. Collective classification problems, as a distinguishing factor, provide another source of input—a relationship graph, defined on the set of entities to be classified. This graph can optionally be directed and/or weighted, and can contain labeled edges that represent different types of relationships. If we are classifying a set of academic papers, one possible relationship graph would be of citations between papers. Another example, if our entities are people, might be email communication frequency between them. In this latter example, we can clearly see how directionality and weighting of network links might be important—after all, two people who exchange hundreds of emails proportionally are likely quite different from the pair who exchanged a single email, or the pair where one sent thirty emails but only received one in return.

It should be clear that the relationship graph is too complex to be easily represented in a fixed-length vector of features, as would be necessary if we wish to directly apply traditional classification techniques. We want to augment our idea of entities as feature vectors so that an entity includes not only its own features, but also makes reference to features from its set of neighbors in the relationship graph, not to mention the directionality and strength of the relationship with each neighbor. The problem is that, in general, entities may have vastly disparate numbers of neighbors, making it futile to try to build a feature vector by concatenating the features of all of an entity’s neighbors. Some previous collective classification techniques have taken the clever approach of summarizing the features across each entity’s neighbors, creating a fixed-length vector of neighbor features that supplements the entity features as a source of input. By applying a function like summation, mode, proportion, etc, to the entity feature-vectors of the set of neighbors, techniques such as the iterated classification algorithm (ICA) are able to shoehorn a lossy version of the relationship graph into a fixed-length feature vector suitable for use with traditional classifiers. Similarly, systems such as kLog [8] use graph kernels to summarize a graph into a fixed-length vector format. When such algorithms include the predicted class of the neighbors in the set of features to be summarized, they need to update these summaries after each new set of class predictions is produced. This inspired the “iterated” part of ICA’s name, since one must build a classifier, update each entity’s neighbor features with the new class predictions, and then build a new classifier for these updated vectors, repeating until the predictions stabilize. Another method called Gibbs sampling (GS) shares the summarization approach with ICA, allowing it to utilize a traditional classifier that produces probability distributions over class labels for each entity. This technique replaces ICA’s iteration step with repeated sampling from an approximate class distribution given the summarized neighbor features, thus enabling prediction of the most likely class for each entity.

Other existing collective classification techniques eschew the traditional classifier approach in favor of statistical models that take relationships into account directly. Loopy belief propagation (LBP) and mean-field relaxation labeling (MFRL) are both approximate inference mechanisms applied to an underlying Markov random field (MRF), which is a model that estimates co-occurrence probabilities of classes with, in this case, both the features of the entity itself and those of its extant neighbors. Such an approach need not summarize neighbor data as in ICA and GS, instead learning probabilities directly tied to individual edges of the relationship graph. The issue with an MRF in the case of collective classification is that it is an undirected model, leaving it to approximate the potentially directed, weighted relationship graph as an undirected, unweighted graph. While these techniques are among the best in practice on some real-world datasets [9], it is easy to see that they would have trouble coping with data where link weight and/or directionality are important predictors. More comprehensive statistical approaches such as Markov Logic Networks (MLNs) [10] require vast computational resources and tend to perform worse than the above approaches in practice on real-world datasets [11].

The idea behind our new technique, recurrent neural collective classification (RNCC), is to create a recurrent neural network capable of learning to utilize all the information pertinent to an entity: its own features, the directed and possibly weighted relationships with its neighbors, and the features of each of those neighbors as well. That each entity has a different number of neighbors need not be a problem here; recurrent neural networks are capable of memory, and so the pertinent features of each neighbor and its corresponding relationship can be presented in a temporal sequence that the network can learn to efficiently encode and aggregate. In essence, the network learns its own summarizing function across neighbor and relationship features, and this function will be automatically selected for optimal prediction on the problem at hand. This is attractive when compared with ICA and GS, where one must often test several summarization approaches to find one best suited to the problem [12]. In contrast to techniques based on MRFs, RNCC captures the weight and directionality of edges by incorporating them directly as inputs.

The existing approaches to collective classification are designed to incorporate information about an entity’s direct
neighbors into the description of that entity so that it might be better classified. But the relationship graphs given as input to such problems contain more information still. So far, these approaches have only considered direct neighbors of an entity, or in other words, the entity’s neighborhood of radius one. For some applications, relationship graph neighborhoods of larger radius may provide critical information. For example, in trying to determine the origin of a product, the purchaser’s link to the retailer is likely to be less useful than the retailer’s links to distributors, even though these are outside the neighborhood of the purchaser. We will refer to relationships like these—where the relationship graph distance between the entities is greater than one—as expanded relational neighborhoods. We will explain in Section III-C how RNCC can learn to utilize expanded relational neighborhoods by creating additional input features from the network’s own learned distributed representations of each entity. But first we present a more technical overview of the technique itself.

III. RECURRENT NEURAL COLLECTIVE CLASSIFICATION

We will begin in Section III-A by presenting an overview of the neural architecture that our technique requires and an example of how it is used, followed by a detailed explanation of how the network is able to represent expanded relational neighborhoods in Section III-B. We then present the full activation and training rules for the network in Section III-C. The technique described in this paper borrows a few general ideas from previous neural network approaches to the more general problem of relational classification (e.g., [13, 14, 15, 16]); this is further discussed in Section V.

A. Neural Architecture

RNCC utilizes a neural network with a specialized structure that allows it to aggregate information about an entity’s neighbors and learn to use that information efficiently to produce better classifications. The network architecture required by RNCC, depicted in Fig. 1, is best explained in stages. Across the top of the figure is a sub-network that represents a non-collective classifier version of RNCC. In fact, this is just a canonical neural network of the long short-term memory (LSTM) variety (see Section III-C) mapping entity features to predicted classes. Here, an entity’s features pass through a hidden layer of self-recurrent memory cells and are used to predict the entity’s class. Things become more interesting when RNCC is set to take advantage of one or more types of neighbor data. Given that we know each entity’s features, as well as the relationship graph that connects the entities, we can provide this information to the network for each neighbor of the entity we want to classify. Since a given entity may have more than one neighbor, and these are generally unordered, we shuffle the neighbors into a random sequence, presenting inputs related to each neighbor on separate steps in a temporal sequence. The network processes each neighbor, integrating the new information with the aggregate representation of all previous neighbors, stored in the memory cell layer labeled Neighbor Data Accumulator. When all neighbors have been processed in this way, the aggregate representation is integrated into the overall representation of the entity that we wish to classify, at the layer labeled Learned Entity Representation, which is subsequently used to predict that entity’s class.

As an example, consider the entities and relationship graph shown on the left in Fig. 2. In this case, our entities are people connected by various types and strengths of relationships; we want to use what we know about each person, together with their relationships, to predict their occupation. Suppose we want to use a previously trained RNCC network to classify Walt, whose neighbors in the provided relationship graph are Gus, Hank, and Jesse. Before presenting Walt’s features to the network, we present and aggregate the neighbor data. To begin, we select a random neighbor from the list, say Hank, and impose details about him on the neighbor input layers as shown on the right in Fig. 2, time-step 1. Specifically, we activate units in the Relationship Details layer that describe Hank’s relationship with Walt; we turn on units corresponding to Hank’s entity features on the Known Neighbor Features input layer; and we activate our most recent prediction of Hank’s class via the Predicted Neighbor Class input layer. The network processes these inputs by passing them through learned weighted connections to the (previously empty) Neighbor Data Accumulator layer, where we now have a representation of Hank that will persist, via the recurrent connections in the memory cells, until the next time-step. We next choose another random neighbor, say Jesse, and present his features to the network in the same way (time-step 2), with this new data modifying the existing representation of Hank on the Neighbor Data Accumulator layer. Subsequently, the final neighbor Gus is presented in the same way (time-step 3), completing the accumulated neighbor representation, which now contains data from all three of Walt’s neighbors. We then present Walt to the network as the entity to be classified by imposing his features on the Known Neighbor Features layer (time-step 4). The network passes this information, and the accumulated neighbor data, on to the Learned Entity Representation layer, completing the network’s representation of Walt, which includes relevant information about all his neighbors. Finally, the network translates this representation into a prediction about Walt’s class, expressed over the output layer labeled Predicted Entity Class. At this point, if we know what Walt’s class should be and want to further train the network, we compare the network’s prediction with the true class and adjust the network’s weights accordingly, as discussed in Section III-C.

The above example provides a basic view of the information flow in RNCC, but the network’s actual operation can be slightly more complex when we wish to use additional neighbor information, as described in the next section.

B. Using Learned Neighbor Representations

The above sources of neighbor information are all utilized by existing collective classification techniques. However, RNCC’s neural network architecture allows it to take advantage of a further source of information that can be helpful in certain contexts.
Fig. 1: Neural network architecture underlying RNCC. Rounded rectangles represent layers of neural units; the left column of these are input layers, the middle column are processing layers built of LSTM memory cells (whose self-recurrence is indicated visually by a circular arc drawn on each layer), and the right column contains the sole output layer. Solid arrows between layers represent matrices of trainable connection weights; in this simplified presentation, each such arrow into a given memory cell layer represents three separate weight matrices—one projecting to the cells themselves, and one each to the input gates and output gates, as shown in Fig. 3. Dashed arrows represent pathways via which network representations may be stored externally for use as supplementary neighbor inputs. The shaded area at the top represents a standard neural network classifier that does not take relationship data into account. The bottom shaded area contains several additional alternate types of input that can be presented to the network for each direct neighbor of the entity being classified; these include the details of the relationship (such as strength, type, and directionality), the neighbor’s features and predicted class, and the network’s most recent learned representation for the neighbor.

Fig. 2: Depiction of an example trial of RNCC. On the left, an example set of entities to classify, connected according to a relationship graph—in this case, people involved in a drug investigation whom we wish to classify according to their role. Solid arrows represent “associates” relationships, while the dashed arrow means “investigates”; arrowheads represent the directionality of the relation, and thickness of an arrow represents relationship strength. The shaded region includes the direct neighbors of the person we are presently classifying, Walt. On the right, a depiction of the RNCC network at each time step during the trial. The RNCC version depicted here is simplified for the purpose of illustration, involving only three of the four possible neighbor input layers; from top to bottom: Relationship Details, Known Neighbor Features, and Predicted Neighbor Class, which is available if we have ever tried to classify the neighbors previously. To classify Walt, we begin by presenting and accumulating information about his direct neighbors in the relationship graph, in a random order. Then we present Walt’s features which, combined with the neighbor data, produces a prediction of Walt’s class.
As explained in the example above, RNCC’s neural network naturally creates a learned representation of each entity being classified; this representation is in the form of an activity pattern distributed across the units in the layer labeled Learned Entity Representation. This representation, however, incorporates encoded information about all of the entity’s neighbors—information fed in through the Neighbor Data Accumulator layer. Thus, the network’s learned representation of Walt includes information about his relationships with his neighbors Gus, Hank, and Jesse; information that is neither present in Walt’s entity features or in the network’s predicted classification for him. Much like the predicted class outputs, the most recent version of this neural representation can be stored for each entity and used as a supplementary neighbor input on the layer labeled Learned Neighbor Representation.

In the example above, the network’s representation of Walt gradually comes to incorporate information about Mike, even though Mike is not presented as an input when Walt is activated. To see how this happens, we must examine how the network’s learned representations of the neighbor, Gus, that connects Walt to Mike. When the network first processes Gus, it integrates information about all of Gus’s neighbors—Walt, Hank, and, crucially, Mike—into a new learned representation for Gus. This representation for Gus remembered outside the network for use as an input later. When the network next processes Walt, and Gus is presented as a neighbor, this saved representation of Gus is one of the inputs. Since this representation includes information about the neighborhood around Gus—which includes Mike—the network’s representation for Walt now also includes information about Mike, by way of Gus. Iterating this idea, each node’s learned representation eventually comes to integrate information about arbitrarily distant nodes in the graph.

While techniques like ICA iterate over predicted classifications, thus potentially allowing a previous class prediction to provide a modicum of information available in expanded relational neighborhoods, this information is limited to the pre-defined space of possible classifications and is highly constrained by training towards the desired outputs. The neighbor representations used by RNCC, by contrast, are arbitrary in size and free to adapt during learning to provide a multidimensional source of information about an entity’s expanded relational neighborhood, making them a powerful addition to the classification process.

When using this new data, each neighbor input now comes with a representation that includes information about its wider role in the relationship graph, instead of just its relationship to the entity currently being classified. This enables a sort of recursive ripple-effect, allowing each entity to have a fuller view of its place among its peers, as follows. The first time we generate a learned representation for a given entity, it includes data only from its direct neighbors; but the second time around, each neighbor knows about its own neighbors, and the central entity’s representation now contains data from its radius-two neighborhood in the relationship graph. As we recalculate and refine these learned representations, each entity’s representation comprises a larger and larger view, until it reaches the far corners of the relationship graph—or until the influence of remote entities is too small to register. This potentially wide view of the network will be beneficial in cases where network properties beyond direct neighbor relationships are important for classification; we demonstrate this in Section V.

C. Neural Network Underpinnings

The neural network underlying RNCC is built using the long short-term memory (LSTM) framework [17]. LSTM networks, like simple recurrent networks (SRNs) [18], are capable of processing sequential inputs and outputs by making use of a recurrent hidden layer. LSTM networks differ from SRNs in two main ways. The first difference is a simplification of SRNs: Whereas an SRN uses a copy-back context layer to maintain state over time, the units in LSTM’s hidden layer—called memory cells (see Fig. 3)—simply carry over their activation level from the previous time-step to the current one. In the context of gradient-descent training techniques such as back-propagation, this unit-level context provides a cleaner way to maintain error signals across temporal sequences, resulting in a superior ability to learn from sequential inputs—a known problem with gradient learning techniques [19]. Since RNCC is highly dependent on learning long sequences when there are many neighbors, LSTM seems an obvious choice over an SRN.

LSTM’s complexity is increased a bit due to its second difference from SRNs. LSTM memory cells have supplementary “gate” cells that can multiplicatively modulate their input and output signals. These input and output gates learn to control their memory cells via the same learning mechanism used by the memory cells themselves, namely gradient descent. While this change does not make LSTM theoretically more powerful—given that a neural network with a single hiddle layer is a universal function approximator—it does give individual memory cells the power to multiply signals as well as add them, which can lead to better outcomes in practice. Another way to conceptualize an input gate is as a secondary source of input for the memory cell, which now takes two inputs and multiplies them, rendering memory cells similar to the sigma-pi units used in higher-order neural networks. Pilot experiments indicate that, without gate units, LSTM networks applied to problems contains short sequences perform little better than SRNs. The addition of input gates improves accuracy markedly, and output gates further improve performance for some problems. While our initial tests showed that output gates could potentially be omitted in RNCC, we include them in the experiments reported here for compatibility with published accounts of LSTM.

To train our LSTM network we use a gradient descent algorithm called LSTM-g [21], which extends the original LSTM training algorithm to allow for multiple layers of memory cells to exist in series. While such multi-layer LSTM networks are sometimes trained with back-propagation thru time (BPTT), LSTM-g is both faster and more localized, lending it greater biological plausibility. For further details about the LSTM architecture and the LSTM-g training algorithm, see [20].

RNCC utilizes memory cells with input gates and output gates, ignoring the later additions of forget gates [21] and
depicted in lowercase and boldface in equations (e.g. a specific architecture using a vector/matrix format. Vectors are simplifying the LSTM-g activation and learning rules for our operation of a general memory cell layer by solving and cells as used by RNCC. In what follows, we describe the peephole connections [22]. Fig. 3 depicts a layer of memory cells as used by RNCC. In what follows, we describe the operation of a general memory cell layer by solving and simplifying the LSTM-g activation and learning rules for our specific architecture using a vector/matrix format. Vectors are depicted in lowercase and boldface (e.g. a) and are column vectors unless explicitly transposed (e.g. aT). Matrices are uppercase and boldface (e.g. A). The symbol ⊙ is used to denote element-wise multiplication between vectors or matrices. Thus, if a = (ai) and b = (bi), then a ⊙ b is a vector of the same length as both inputs, where each element is (aiibi); similarly for matrices, given A = (aij) and B = (bij), A ⊙ B = (aijbij). In the following, x vectors represent net inputs, s vectors represent retained states, and y vectors represent unit activations that have been passed along with their connective matrices, a memory cell may have more than one of each.

When activating a memory cell layer (see Fig. 3), we begin by calculating the net input x, for the input gates by multiplying the incoming weight matrix Wc with the activation vector yin of an upstream layer. We subsequently apply the logistic function f to calculate the input gate activation vector yi.

\[ x_i = W_c y_{in} \]  
\[ y_i = f(x_i) \]

The activation of the memory cell itself is similar but a bit more complex. The net input \( x_c \) proceeds analogously until the input gates act, attenuating it via element-wise multiplication with the input gate activations \( y_i \). The memory cell maintains a state across time, so its new state \( s_c \) is the sum of its old state \( s_c \) and the gated net input.

\[ x_c = W_c y_{in} \]  
\[ s_c = s_c + x_c \odot y_i \]  
\[ y_c = f(s_c) \]

Next the output gate is activated, analogously to the input gate. This activation \( y_\omega \) is used to gate the memory cell activation \( y_c \) and produce the memory cell’s final output value \( y_{out} \), which can then be passed on to downstream layers. This concludes the activation phase of a memory cell layer.

\[ x_\omega = W_\omega y_{in} \]  
\[ y_\omega = f(x_\omega) \]  
\[ y_{out} = y_c \odot y_\omega \]

To train the weights, we perform back-propagation to determine each unit’s error responsibility \( \delta \). By combining these responsibilities with each weighted connection’s eligibility \( \varepsilon \)—a measure of how much the particular connection has contributed to the receiving unit’s activation during this trial—we can compute an appropriate weight change for each connection. In the following, E matrices represent per-connection eligibility traces, and \( \delta \) vectors are per-unit error responsibilities.

Immediately after calculating the activation \( y \) for each pathway above, we must also update the eligibility for all connections along that pathway to record their contributions to the current activation state of the network. These formulas are derived using the gradient of the network error, working backwards from each unit to the corresponding weighted connection (see [20] for more information). The memory cell and any upstream pathways (here, the input gates) utilize aggregate eligibility traces that reflect each weight’s total contribution to the memory cell’s state \( s_c \). These traces take into account the gradient of the computational pathway between the weights and the memory cell state as its \( \rho \) value. On the other hand, the output gate eligibilities, which do not affect the memory cell state, are more straightforward: \( \rho_\omega \) is simply a column vector of ones, which is multiplied with the transposed input vector to create an appropriate eligibility matrix that reflects, for each connection, the most recent input to that connection.

\[ \rho_s = x_c \odot f'(x_c) \]  
\[ \rho_i = y_i \]  
\[ \rho_\omega = 1 \]  
\[ E_s = E_c + \rho_s y_{in}^T \]  
\[ E_i = E_c + \rho_i y_{in}^T \]  
\[ E_\omega = \rho_\omega y_{in}^T \]
When a training target is presented to the network, error responsibilities are calculated for the output units (see (23)) and propagated backwards. Since we are examining a general memory cell layer that could be anywhere in the network, it will suffice to know that the error responsibilities $\delta_{\text{out}}$ for the layer immediately downstream have been calculated. By multiplying the transposed weight matrix with this vector, we get the error responsibility vector $\delta_{\text{in}}$ for our layer. Using the gradient, we work backwards through the operations performed by the network, calculating the error responsibility for the output gates ($\delta_{\omega}$), memory cells ($\delta_{c}$), and input gates ($\delta_{t}$).

$$\delta_{\text{in}} = W^T_{\text{out}} \delta_{\text{out}}$$  \hspace{1cm} (15)

$$\delta_{\omega} = f'(x_{\omega}) \odot y_{\omega} \odot \delta_{\text{in}}$$  \hspace{1cm} (16)

$$\delta_{c} = \delta_{t} = f'(s_{c}) \odot y_{\omega} \odot \delta_{\text{in}}$$  \hspace{1cm} (17)

We expand the per-unit $\delta$ vectors into per-connection error responsibility matrices $D$ by multiplying them with a transposed vector of ones. Finally, we can calculate the weight change matrices $\Delta W$ by combining the scalar learning rate $\alpha$ with the eligibility traces and error responsibilities.

$$D_{\lambda} = \delta_\lambda 1^T$$ for $\lambda$ in $\{t, c, \omega\}$  \hspace{1cm} (18)

$$\Delta W_{\lambda} = \alpha (E_{\lambda} \odot D_{\lambda})$$ for $\lambda$ in $\{t, c, \omega\}$  \hspace{1cm} (19)

If we want to continue to propagate error information backwards from any of these layers, we have slight problem: The $\delta$ values defined above do not reflect the gradient of the network structure prior to the memory cell states, where applicable. To remedy this, we instead pass back $\delta'$, which completes the error responsibility for each node as the product of each unit’s $\delta$—the error gradient after the memory cell state—and its $\rho$—the gradient before the memory cell state:

$$\delta'_{\lambda} = \delta_{\lambda} \odot \rho_{\lambda}$$ for $\lambda$ in $\{t, c, \omega\}$  \hspace{1cm} (20)

The neural architecture underlying RNCC contains two layers of the memory cells described above. Each such layer may have multiple inputs from upstream layers, in which case each upstream layer projects its own set of three weight matrices forward to the memory cells.

The architecture’s only other non-input layer is the output units $\theta$, which are stateless neural units like the input and output gates. The architecture uses the cross-entropy error function to calculate the responsibilities of the output units with respect to the target vector $t$, giving a simple $\delta_{\theta}$.

$$x_{\theta} = W_{\theta} y_{\text{in}}$$ \hspace{1cm} (21)

$$y_{\theta} = f(x_{\theta})$$ \hspace{1cm} (22)

$$\delta_{\theta} = t - y_{\theta}$$ \hspace{1cm} (23)

The eligibility traces, delta matrix, and weight changes for the output units are simple, resembling those of the output gates in a memory cell layer.

$$E_{\theta} = 1 y_{\text{in}}^T$$ \hspace{1cm} (24)

$$D_{\theta} = \delta_{\theta} 1^T$$ \hspace{1cm} (25)

$$\Delta W_{\theta} = \alpha (E_{\theta} \odot D_{\theta})$$ \hspace{1cm} (26)

A detailed derivation of these learning rules and a proof that they work towards minimizing the observed error can be found in [20].

Our RNCC implementation was built using an open-source, Java-based neural network toolkit called XLBP. Standing for eXtensible Localized Back-Propagation, this framework can be used to easily construct second-order neural networks (i.e., those in which neural units are capable of both addition and multiplication of their inputs) including those of the LSTM paradigm, and train the resulting networks using the LSTM-g algorithm described above. Source code for both XLBP and RNCC are available via the first author’s website.

IV. EXPERIMENTS AND RESULTS

We evaluated our RNCC technique on both synthetic and real-world datasets. The synthetic datasets were designed to test the technique’s ability to make use of both direct-neighbor information and, later, information present only in expanded neighborhoods around each entity. The real-world datasets provide an avenue for comparing RNCC with the published results of other collective classification techniques.

A. Synthetic Datasets

We first tested RNCC on synthetic datasets generated to showcase its ability to take advantage of both direct-neighbor and expanded relational neighborhood connections when making classifications. The idea underlying the synthetic datasets is that each node in the relationship graph has a base color—red or blue—provided as an input feature, and by default should be classified as this same color. Blue nodes, however, transfer some of their blue-ness to their surrounding neighbors, rendering purple any previously red nodes that happen to be reachable by a directed path of a defined length from a blue node (see Fig. 4). Thus, classification methods that do not account for relationship data will fail to reliably identify purple nodes; further, those that do not take expanded relational neighborhoods into account will suffer when the range of a blue node’s influence is 2 or larger.

More concretely, each entity in our synthetic datasets has two mutually exclusive binary input features that represent red and blue; we will refer to these as red-feature nodes and blue-features nodes. Our desire is to classify each node into one of three classes: red, blue, or purple. Blue-feature nodes are always blue-class as well, and a red-feature node can be thought of as red-class by default, unless it is too close to a blue-feature node, in which case its class becomes purple. The definition of “too close” depends on the smallest neighborhood radius around the red-feature node that contains a blue-feature node, and this radius will vary between datasets. For example, in a dataset with radius 1, all red nodes that have a direct blue-feature neighbor will belong to the purple class instead of the red class. Similarly, in a dataset with radius 2, any red-feature node connected to a blue-feature node via a path of length 2 or less has its class turned purple.

We tested RNCC on three synthetic datasets generated as specified above, each with a different radius—we will call the

\[http://www.cs.umd.edu/~dmonner/\]
datasets R1, R2, and R3 respectively. Each dataset contains 1000 nodes, and in each, blue-feature nodes made up only 5% of the population. We adjusted the link density in the relationship graph to keep an approximately stable population of purple-class nodes. The R1 dataset had a link density of 0.01—meaning that 1% of possible links were present—leading a node to have an average of 10.25 neighbors in practice. Of R1’s population of 1000 nodes, 46 were generated with the blue-feature, and with the blue-influence radius set at 1, this led to 358 purple-class nodes and 596 red-class nodes. For R2, exactly 50 nodes happened to be set to the blue-feature, and with our link density set at 0.003, we ended up with 396 purple-class nodes and 554 red-class ones; the average node had a two-neighborhood consisting of 11.85 other nodes. Dataset R3 had only 43 blue nodes and a link density of only 0.002, but this led to a three-neighborhood size of 15.00 on average, with 348 purple nodes to complement the 609 red ones.

The results of running several RNCC variants on these three datasets are shown in Fig. 3. We tested each dataset using a variety of RNCC configurations in which the underlying neural network either was or was not allowed to take advantage of each type of neighbor data available—neighbor features (F), predicted neighbor classes (C), and/or the previously learned neural representations of each neighbor (R; see Section II-B), resulting in 8 possible configurations. The leftmost configuration for each dataset utilizes no neighbor data at all and functions as a control; in other words, it is a traditional non-collective classifier—in this case, a canonical LSTM neural network in which the recurrent self-connections are never utilized—used to comparatively demonstrate the benefit of using neighbor data.

The input and output layer sizes of these networks are defined by the parameters of the dataset. In this case the feature input layers have 2 units while the class-related input and output layers each consisted of 3 units. The Relationship Details input layer had 2 units, representing the connection strength in each direction; since the input graphs were unweighted, each of these nodes always took either 0 or 1 as its value, depending on whether a directed connection was present. Both internal memory cell layers were set to a size of 20 units. Weighted connections between pairs of units in connected layers (where solid arrows exist in Fig. 1) are present with a probability of 0.8. Thus, a typical RNCC network with all types of neighbor input enabled has approximately 2400 trainable weights.

We ran 20 individual RNCC networks, each initialized with random starting weights, for each combination of dataset and network configuration (i.e., for each boxplot in Fig. 5). For each such combination, we report the classification accuracy

![Fig. 4: Example classifications for the red/blue synthetic dataset in a small graph. The darkly shaded node is the only blue node in the graph, while all other nodes have the red input feature and should be classified as red by default. Nodes are labeled with their directed distance from the central blue node. In this example, the radius of blue influence is two, so any red-feature nodes that can be reached from the blue node via path of length two or less—lightly shaded in this figure—are classified as purple instead of red.](image)

![Fig. 5: Performance comparison of RNCC variants on the synthetic red/blue task using three different radius parameters. RNCC variants are defined by the neighbor data that they utilize, starting with none (leftmost boxplot) and progressively adding and combining neighbor features (F), predicted classes (C), and learned representations (R). For each dataset, the boxplots show the performance distribution of 20 runs of each RNCC variant. The center line of a boxplot shows the distribution median, and the boxed-in portions show quartiles above and below the mean; the vertical whiskers show the extent of the rest of the distribution up to 1.5 times the interquartile range, and any outliers are shown as individual points beyond the whiskers. Horizontal dashed lines indicate the level of performance of the simplistic strategy of (correctly) classifying all blue-feature nodes as blue-class, and (sometimes correctly) classifying all red-feature nodes as red-class, making no purple classifications at all. The graphs show that, for datasets with radius greater than 1, RNCC’s use of learned neighbor representations—which encode information beyond their direct neighbors—significantly increases accuracy.](image)
on a 10-fold cross-validation task where folds were of equal size and entity assignments to each fold were random. Relationship information between folds was retained during both training and testing, and, for classifiers that utilize neighbor class information, ground truth classes were available for entities assigned to folds not under test, following standard practice. Networks were allowed to train on each fold for a maximum of 25 epochs, where an epoch is a single randomized pass through the entire training set. The learning rate was 0.1. Unit activations in the network are reset after each trial, and the network’s class predictions and learned representations are reset between folds. A network’s prediction for a test trial is defined as its most active output unit at the conclusion of that trial, and accuracy is defined as the fraction of correct predictions.

Examining the results for dataset R1 (top panel in Fig. 5), we see that the non-collective version of RNCC (the leftmost boxplot) performed the worst. In fact, since it had no reliable information on which to base purple-class assignments for red-feature nodes, the strategy it learned was to essentially assign the red class to all such nodes, since it is more prominent than the purple class. Thus, in the best case it learns to classify all of the red and blue nodes—the trivial ones in this example, 642 of 1000 data points—correctly, while all the purple-class nodes are misclassified as red. This level of performance is indicated in the plot by a horizontal dashed line. Adding any relationship information—features (F), predicted classes (C), or learned representations (R)—dramatically improved performance, causing all such classifiers to routinely solve the task perfectly.

Moving on to the more difficult R2 dataset (middle panel in Fig. 5), we see again that the leftmost non-collective classifier performs at the level predicted, where it does no better than it would by never assigning the purple class to any node. While adding neighbor feature and/or class information in the next three boxplots produces a significant accuracy improvement, these classifiers do not come close to full accuracy. This is because the purple class is no longer fully predicted by direct neighbors—we can correctly classify purple entities that have direct blue neighbors, but we miss purple entities that have a distance-two relationship to a blue entity. Adding the RNCC network’s learned representations as a neighbor input solves this problem, since these representations can learn to encode information about each entity’s extended neighborhood. RNCC variants that used learned neighbor representations as inputs often achieved accuracies over 95%.

Finally, the results for the R3 dataset (bottom panel in Fig. 5) are similar, showing that RNCC can capture enough information about neighbors at distance 3 to achieve accuracies exceeding 90%. One notable difference between R2 and R3 is that direct-neighbor predicted class information seems to have become more predictive for R3, which is initially surprising given that R3 relies less on direct neighborhood relationships than either previous dataset. This is explained by a deeper look at the structure of the data, however, as shown in Fig. 5. In R2, if we observe a node $A$ to have a purple-class neighbor $B$, there are two possibilities for how $B$ became purple-class—either $B$ is directly connected to a blue node, or $B$ has a length-two path to a blue node. In the former case, $A$ would also need to be purple, but in the latter case we gain no information about $A$. In essence, given one purple neighbor, $A$ has an even chance of being purple itself in R2. The chance that $A$ is purple will increase slowly with each additional purple neighbor it has. Switching over to dataset R3, there are now three possibilities for how $A$’s neighbor $B$ gained its purple status, and two of these directly imply that $A$ is purple too. Thus, a single purple neighbor gives $A$ a two-thirds chance of being purple, lending purple neighbors more predictive power in R3. This is, of course, an oversimplification—the predictive power of purple neighbors will be skewed by the frequency with which each case occurs—but we can see that as we increase the radius further in our task, neighbor classes gain more and more predictive power. However, this behavior is closely linked to the structure of our synthetic task, and there is no reason to expect neighbor class information to gain similar predictive power with distance in real world datasets. It is worth noting that, even in R3 where neighbor classes are highly predictive, using the network’s learned representations of neighbors as input provides better predictions still. This demonstrates that RNCC’s neural network learns to encode useful information about an entity’s expanded relational neighborhood in its representation for that entity, which can then be used effectively during the classification.

### B. Real-world Datasets

We subsequently tested RNCC on real-world datasets that have been used to demonstrate the effectiveness of previous collective classification algorithms. We first pitted the variants of RNCC outlined above against each other, and then compared the full version of the algorithm to the best published results of the most popular collective classification approaches. We used four publicly available datasets, as follows:

1. The Cora dataset [23] is a collection of academic papers in the machine learning field that are grouped into seven
topic classes. The collection contains 2,708 papers, each having 1,433 binary features based on the occurrence of important words in the paper itself. The 5,429 directed links in this dataset represent citations. Following the methodology used to generate best reported results on this dataset [8], our experiments here used a 10-fold random cross-validation scheme.

2) The Citeseer dataset [24] is another group of 3312 academic papers, grouped into six topic classes. Each paper has 3703 binary features, again based on word occurrence, and the 4732 links represent citations as before. Again following [9], we test this dataset using 10-fold cross-validation.

3) The WebKB dataset [25] is a collection of 877 academic web pages to be classified as one of five web page types. Each page has 1703 binary features, again representing the presence or absence of individual words. The 1608 directed links define the hyperlink structure among the web pages. Interestingly, this corpus is drawn from four different universities, and so the hyperlink graph contains four disconnected subcomponents. We adhere to the testing principles established by [26], performing 4-fold cross-validation, where each fold consists of one of these disconnected subcomponents.

4) The Terrorist Relationships dataset [27] is a collection of 851 relationships among suspected terrorists, to be classified into one of four groups such as family or accomplice. Each relationship has 1224 binary features based on keywords and tags from the profiles of the suspects involved. Relationships form a graph with edges connecting relationships that share an individual. Following the original work, we utilize two-fold cross-validation on this dataset. The publicly available version of this dataset appears to correspond to the single-label classification task, whose results appear in Fig. 3 of [27]; the dataset also appears to be missing 32 relationships compared to what was described in the paper.

Table I compares the performance of variants of RNCC on these datasets, where as before the variants are based on the presence or absence of each type of neighbor input data. Each row of this table was again generated from 20 individual runs of randomly initialized RNCC networks. The network parameters and layer sizes are as described Section IV-A, except that input and output layer sizes change according to the number of features and classes defined by each dataset. The conclusions we can draw from Table I are that neighbor features and classes are both salient information sources for these real-world datasets, and that the learned neighbor representations provide significant additional information on at least the WebKB dataset. In comparing RNCC to other classification methods in what follows, we use only the full version of RNCC that utilizes neighbor features, classes, and representations, even if that variant did not produce the best score for a given dataset.

We compared RNCC against a variety of popular methods for collective classification and, for context, we also compared a traditional LSTM network—essentially, RNCC stripped of any neighbor input data—against non-collective methods applied to the same datasets. The only non-collective classification results we encountered on these datasets were from logistic regression (LR) models—sometimes referred to as maximum entropy models—and naïve Bayes (NB) models. The collective classifiers with published results available included the iterated classification algorithm (ICA) using either logistic regression or naïve Bayes model as its internal classifier; Gibbs sampling (GS) using either of the two mentioned internal classifiers; or a Markov random field model where approximate inference is performed by either loopy belief propagation (LBP) or mean-field relaxation labeling (MFRL).

Table 1 compares RNCC’s performance with the best known published results of these other approaches on our test datasets. The papers in which these other results were published generally do not specify if they are average-case or best-case numbers; since many of these techniques are deterministic, it makes sense to compare them to both RNCC’s best-case performance and its average performance over all 20 runs. Looking only at RNCC’s average-case numbers, it manages to outperform all alternate approaches surveyed on all datasets, except for a single near-tie with LBP on the Citeseer dataset. RNCC’s best-case results, meanwhile, outperform the reported results for all other approaches on every dataset, often by a substantial margin.

\begin{table}[h]
\centering
\caption{Performance comparison of RNCC variants on real-world data}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\textbf{Dataset} & \textbf{Neighbor Data} & \textbf{Feat} & \textbf{Class} & \textbf{Repr} & \textbf{Mean} & \textbf{SD} & \textbf{Max} \\
\hline
 & & & & & & & & & \\
Cora & - & - & - & & 0.7728 & 0.0032 & 0.7792 \\
 & + & - & - & & 0.8702 & 0.0035 & 0.8763 \\
 & - & + & - & & 0.8832 & 0.0033 & 0.8900 \\
 & - & - & + & & 0.8765 & 0.0033 & 0.8818 \\
 & + & + & - & & \textbf{0.8941} & \textbf{0.0036} & \textbf{0.8996} \\
 & - & + & + & & 0.8929 & 0.0026 & 0.8973 \\
 & - & + & + & & 0.8877 & 0.0029 & 0.8911 \\
 & + & + & + & & \textbf{0.8960} & \textbf{0.0027} & \textbf{0.9010} \\
\hline
WebKB & - & - & - & & 0.7274 & 0.0033 & 0.7349 \\
 & + & - & - & & 0.7572 & 0.0027 & 0.7627 \\
 & - & + & - & & 0.7599 & 0.0037 & 0.7705 \\
 & - & - & + & & 0.7480 & 0.0031 & 0.7545 \\
 & + & + & - & & \textbf{0.7746} & \textbf{0.0031} & \textbf{0.7835} \\
 & - & + & + & & 0.7671 & 0.0030 & 0.7739 \\
 & - & + & + & & 0.7657 & 0.0031 & 0.7717 \\
 & + & + & + & & \textbf{0.7758} & \textbf{0.0034} & \textbf{0.7811} \\
\hline
Terrorists & - & - & - & & 0.8404 & 0.0063 & 0.8552 \\
 & + & - & - & & 0.8478 & 0.0064 & 0.8587 \\
 & - & + & - & & 0.8417 & 0.0059 & 0.8506 \\
 & - & - & + & & 0.8424 & 0.0064 & 0.8563 \\
 & + & + & - & & 0.8479 & 0.0076 & 0.8597 \\
 & + & + & + & & \textbf{0.8508} & \textbf{0.0085} & \textbf{0.8620} \\
 & + & + & + & & 0.8438 & 0.0073 & 0.8552 \\
 & + & + & + & & \textbf{0.8528} & \textbf{0.0066} & \textbf{0.8632} \\
\hline
\end{tabular}
\end{table}

\footnote{Since the original source did not report these numbers directly, they were estimated (rounded up) from the bars labeled “All” in Figure 3 of [27], with the LR row corresponding to the figure’s “Flat Model” and LBP corresponding to “RMN Dyad.”}
TABLE II: Performance of RNCC vs. published results of other algorithms

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Neighbors</th>
<th>Algorithm</th>
<th>Accuracy</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cora</td>
<td>No</td>
<td>NB</td>
<td>0.7776</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LR</td>
<td>0.7695</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (avg)</td>
<td>0.7728</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (best)</td>
<td>0.7792</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>NB + ICA</td>
<td>0.8475</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LR + ICA</td>
<td>0.8796</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NB + GS</td>
<td>0.8404</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LR + GS</td>
<td>0.8810</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LBP</td>
<td>0.8766</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MFRL</td>
<td>0.8836</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (avg)</td>
<td>0.8960</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (best)</td>
<td>0.9010</td>
<td></td>
</tr>
<tr>
<td>Citeese</td>
<td>No</td>
<td>NB</td>
<td>0.7587</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LR</td>
<td>0.7321</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (avg)</td>
<td>0.7274</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (best)</td>
<td>0.7349</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>NB + ICA</td>
<td>0.7883</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LR + ICA</td>
<td>0.7732</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NB + GS</td>
<td>0.7680</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LR + GS</td>
<td>0.7699</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LBP</td>
<td>0.7759</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MFRL</td>
<td>0.7732</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (avg)</td>
<td>0.7758</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (best)</td>
<td>0.7811</td>
<td></td>
</tr>
<tr>
<td>WebKB</td>
<td>No</td>
<td>LR</td>
<td>0.8169</td>
<td>[40]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (avg)</td>
<td>0.8404</td>
<td>[40]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (best)</td>
<td>0.8552</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>LR + ICA</td>
<td>0.8439</td>
<td>[40]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LBP</td>
<td>0.8442</td>
<td>[40]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MFRL</td>
<td>0.8440</td>
<td>[40]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (avg)</td>
<td>0.8528</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (best)</td>
<td>0.8632</td>
<td></td>
</tr>
<tr>
<td>Terroris</td>
<td>No</td>
<td>LR</td>
<td>0.757</td>
<td>[42]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (avg)</td>
<td>0.8193</td>
<td>[42]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM (best)</td>
<td>0.8355</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>LBP</td>
<td>0.824</td>
<td>[42]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (avg)</td>
<td>0.8931</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNCC (best)</td>
<td>0.9048</td>
<td></td>
</tr>
</tbody>
</table>

V. DISCUSSION

This paper has presented a novel technique based on recurrent neural networks for the problem of classification with relationships between entities. Dubbed recurrent neural collective classification (RNCC), the technique uses data about each direct neighbor of an entity—a neighbor’s features, predicted class, and learned neural representation—to improve its classification of that entity.

Our comparative results with RNCC show promise. This combination of the LSTM architecture with unsummarized neighbor and relationship data combines the advantages of previous algorithms—the ability of ICA and GS to handle directed relations, and LBP and MFRL’s ability to reason without summarizing the relationship graph—to create a new technique that produces superior results. The novel addition of the network’s own learned neighbor representations to the pool of input data can lead to improved performance in cases where expanded relationship neighborhoods contain information relevant to the desired classification.

The problem solved by RNCC—collective classification—is a specialization of the more general problem of relational classification, where the entities to be classified may be of different types (i.e., having different individual feature-spaces) and be related by various types of relationships—including subcomponent relationships—that can be supported by relational databases. There are existing neural network techniques for relational classification, some of which share features with RNCC. Recursive neural networks (RecNNs) have evolved to process relational classification problems involving directed positional (edge-ordered) acyclic graphs (DPAGs) to general directed acyclic graphs (DAGs) in each case by building a neural network that embeds the relationship graph in question into the neural architecture. The technique was further extended to handle general directed graphs by mapping them to trees. A further extension, called graph neural networks (GNNs), processes most types of graphs without modification. Other researchers have adapted the approach to use the cascade correlation learning method. Ignoring differences in description, the general technique behind these approaches is similar to another technique called relational neural networks (RelNNs). Where RecNNs and GNNs use an explicit graph embedded in the neural network and a settle-and-train learning procedure, RelNNs preprocess the graph into multiple entity-plus-neighbor sets to be presented sequentially and can use traditional backpropagation learning. Despite these apparent differences, both techniques essentially learn a function by which each node processes information from its neighbors, and another by which a node becomes an output value.

RNCC resembles all of these techniques, but differs from them in a number of important ways. RecNNs and GNNs use an explicit copy of the relationship graph to propagate neighbor information, while RelNNs use a Jordan-style simple recurrent network (SRN) to aggregate the neighbor data temporally. RNCC is most similar to RelNNs in this regard, but instead uses the LSTM architecture, which has been shown to be more effective in cases where there are more than a handful of neighbors to aggregate. Second, RNCC takes advantage of several additional sources of input compared to the above techniques: None make use of previously predicted classes of neighbors, and RelNNs have no notion of using the previously learned compressed representations for each neighbor. As we have shown, both types of information can lead to dramatic performance improvements.

RNCC, as formulated here, can only be used for collective classification, rather than the more general domain of the above-discussed relational classification techniques. Because of this, we are unable to present direct performance comparisons of RNCC with these other techniques. We are planning future extensions of RNCC that will be applicable to the more general problem of relational classification, and thus comparable to previous neural approaches.

ACKNOWLEDGMENT

This work was supported by Intelligence Community Post-doctoral Research Fellowship Program award number #2011-1107140011.

REFERENCES


