Solving the Loading Problem for Recurrent Neural Networks with the Highest-Value Walk Problem

Justin Fletcher, *Member, IEEE*

***Abstract*—The abstract goes here.**

***Index Terms*—Recurrent neural networks, Graph theory.**

I. Introduction

RECURRENT neural networks (RNN) are neural networks for which the underlying graph is weighted, directed, and cyclic. As with all neural networks, the purpose of an RNN is to map an input pattern to a desired output pattern. Assuming the topology and transfer function of RNN are fixed, the accuracy of this mapping is dependent only on the synaptic weights of the network, which cooerespond to the edge weights in the underlying graph. The problem of selecting edge weights, such that the difference between computed and desired output patterns is minimized, is known as the *loading problem* [1]. The process of searching for a synaptic weight configuration which solves the loading problem is known as *training* the neural network.

Several procedures exist for RNN training. The inspiration for the presented method is the error back-propagation (BP) algorithm for feed-forward neural network (FFNN) weight selection. In BP, the error produced by the network for a given set of input-output pattern pairs is propagated backward through the network, and a share of the credit for the produced error is allocated to each of the weights according to its responsibility for the error. This is called the *credit assignment problem*. It is possible to assign credit in recurrent neural network using BP Through Time (BPTT) [2], [3], though this approach is known to be computationally expensive, and is prone to convergence to local minima [4]. A novel method for assigning credit for RNN output error is proposed in this work.

II. Approach

The output error of an RNN is a linear combination of error terms, each corresponding to an element of the desired output pattern. Thus, to minimize the total error we must minimize the error of each output vector element. Observe that the output signal from a given neuron is the sum of weighted signals into that neuron, transformed by the activation function. If we take the activation function to be fixed, and the input signals to be incidental to the operation of the network, then we find that the weight is the only feature of the network to which we may assign blame for the produced error. Thus, if a neuron outputs a signal which results in an error with a particular direction (positive or negative), then that error must be induced by those weights which are of the same direction. Further, the larger the error-inducing weight, the more it contributes to the error. Thus, by modifying the error-inducing weight, the error may be reduced. However, this procedure only applies to those weights for which the afferent neuron has an explicitly-defined desired output value. In order to apply this procedure to all weight in the network, a mechanism for defining the error contributions of other weights must be established. Applying inductive reasoning backward through the network, we observe that the maximum credit for the error will always be assigned to the largest weight with the same sign as the error, at each neuron. However, because the underlying graph contains cycles, there is no obvious way to conclude this induction.

An insight from the RNN loading problem domain is required to proceed. First, we define a *propagation* to be the update of the output signals for all neurons in the network. A finite-time RNN produces an output vector after a fixed number of propagations, a quantity which we label L. Thus, only those walks, taken by an input signal, from an input neuron u to an output neuron v, of length L contribute to the output error at v, or v-error $\mathcal{E}(v)$. This walk-length constraint is sufficient to bound the induction over the structure of the underlying graph; we know that the credit assignment path for a given output neuron v must end at v, begin at an input neuron u, and contain L edge traversals. Thus, applying the inductive reasoning above, the weights of the edges which lie along optimum-value (maximum for positive v-error, minimum for negative v-error) walks from each input neuron to a particular output neuron constitute the greatest contributions to the v-error. As such, these weights should be reduced proportionally to the v-error. Repeating this procedure will reduce the output error of the network. We call the problem of finding the optimum-value walk of length L the optimum-value walk problem (OWP). Solving the OWP on the underlying graph of an RNN yields a set of weights which, if reduced, will reduce the error of the RNN.

A. Optimum-Value Walk Problem Specification

Consider a directed, weight graph $G = (V, E)$. In G, we identify a set of input vertices $I \subseteq V(G)$ and output vertices $O \subseteq V(G)$. I and O may, or may not, be disjoint. Given an output vertex $v \in O$, an input vertex $u \in I$, and an optimization direction, we wish to find a walk W such that the value of the walk is optimized. We define the value of a walk $w(W)$ to be

$$w(W) = \sum_{e \in E(W)} w(e), \tag{1}$$

J. Fletcher is with the Air Force Institute of Technology, WPAFB, OH 45433 USA. email: justin.fletcher.8@us.af.mil

where $E(W)$ is the multiset comprising the edges traversed in W. Solving the OWP for all $u \in I$ and $v \in O$ yields, for each v, the set of I, v-walks of optimum value.

B. Mapping OWP to the RNN Loading Problem

Solving the OWP on the underlying graph of an RNN provides a means to solve the loading problem. In this section, a mapping which integrates the two problems is described. The input neurons of the network are mapped to I, while the output neurons are mapped to O. We present an input pattern to the input neurons of the RNN, and perform L propagations. For each $v \in O$, we compute the v-error $\mathcal{E}(v)$, and solve the OWP beginning at every $u \in I$, ending at v, and optimizing such that the value of the path is maximized if $\mathcal{E}(v) \geq 0$ and is minimized otherwise. This procedure yields, for each v, a set $\hat{W}$, which contains the optimal-value walks from each $u \in I$ to v.

$\hat{W}$, the set of comprising the optimal walks for each $u \in I$ to v, may be used to determine how to modify the weights. The credit for the $\mathcal{E}(v)$ assigned to a walk W is proportional its optimal value, relative to the optimal value of other walks. Likewise, the credit assigned to a particular edge $e \in W$ is proportional to $w(e)$, relative to the weights of other edges in W. Thus, we propose the weight-update rule

$$w(e) \leftarrow w(e) - \frac{w(e)}{w(W)} \frac{w(W)}{w(\hat{W})} \mathcal{E}(v), \quad \forall e \in E(W), \quad \forall W \in \hat{W},$$

which simplifies to

$$w(e) \leftarrow w(e) \left(1 - \frac{\mathcal{E}(v)}{w(\hat{W})}\right), \quad \forall e \in E(W), \forall W \in \hat{W}, \tag{2}$$

where

$$w(\hat{W}) = \sum_{W \in \hat{W}} w(W). \tag{3}$$

The description of the OWP and the weight update rule given by Eq. (2) are sufficient to construct an algorithm which utilizes the OWP to search for solutions to the RNN loading problem. Several additional notation conventions must first be specified. ω is a weight matrix, defined such that

$$\omega_{ij} = w(ij) \qquad \forall i, j \in V(G). \tag{4}$$

X is a data set, and is defined such that each element $x \in X$ is an ordered pair of the form

$$x = (\chi, \lambda) = \{\chi, \{\chi, \lambda\}\}$$

where χ and λ are independent sets of real numbers. $\varphi_L(\omega, \chi)$ is the propagation function, which presents the pattern χ into a neural network with weights defined by ω, propagates L times, and returns the output pattern. The following algorithm will adjust the weights of the network, using the described weight update procedure, once for each observation in the data set.

```
1:  procedure OWPTRAINRNN($G$, $I$, $O$, $L$, $X$)
2:      for $(\chi, \lambda) \in X$ do              ▷ Iterate over each observation.
3:          $\mathcal{E} \leftarrow (\varphi_L(G, \chi) - \lambda)^2$   ▷ Compute the error vector.
4:          for $v \in O$ do                     ▷ Iterate over output neurons.
5:              for $u \in I$ do                 ▷ Iterate over input neurons.
6:                  $W \leftarrow$ solveOWP($G$, $u$, $v$, $\mathcal{E}$)
7:                  for $\{ij\} \in W$ do        ▷ Iterate over $e \in W$.
8:                      $\omega_{ij} \leftarrow w(e)\left(1 - \frac{\mathcal{E}(v)}{w(\hat{W})}\right)$ ▷ Update $\omega_{ij}$.
9:                  end for
10:             end for
11:         end for
12:     end for
13:     return ($\omega$)
14: end procedure
```

C. OWP Solution Methods and Analysis

The HWP is solved via depth-first search with backtracking. The root of the search tree is the tail of the walk, u. u may be adjacent to, at most, every other neuron in the network, thus there are $n = |V(G)|$ possible edge traversals. At the next level of the search tree, each search tree node again begets $n-1$ nodes, thereby comprising a set of $(n-1)^2$ nodes. This tree growth repeats $L-1$ times. To construct the Lth layer, all vertices must follow an edge to vertex v. The underlying graph of the network is simple and fully connected, so the Lth layer has $(n-1)^{L-1}$ nodes, as each node must result in a walk which ends at v. Thus, the total number of nodes which must be searched is

$$(n-1)^{L-1} + \sum_{l=0}^{L-1} (n-1)^l. \tag{5}$$

Therefore, the running time complexity of computing an exact solution to the HWP using DFS/BT is $\mathcal{O}(n^L)$.

III. Conclusion

The conclusion goes here.

Acknowledgment

The author would like to thank Dr. D. Mixon for a thorough introduction to graph theory, without which this work would not been possible, and for many fruitful discussions.

References

[1] M. Gori and A. Sperdut, "The loading problem for recursive neural networks," *Neural Networks*, vol. 18, no. 8, pp. 1064 – 1079, 2005, neural Networks and Kernel Methods for Structured Domains. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608005001668

[2] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, vol. 1, no. 4, pp. 339 – 356, 1988. [Online]. Available: http://www.sciencedirect.com/science/article/pii/089360808890007X

[3] M. C. Mozer, "A focused backpropagation algorithm for temporal pattern recognition," *Complex Systems*, vol. 3, pp. 349–381, 1989.

[4] M. Cuellar, M. Delgado, and M. Pegalajar, *Enterprise Information Systems VII*. Dordrecht: Springer Netherlands, 2006, ch. AN APPLICATION OF NON-LINEAR PROGRAMMING TO TRAIN RECURRENT NEURAL NETWORKS IN TIME SERIES PREDICTION PROBLEMS, pp. 95–102. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-5347-4-11

PLACE
PHOTO
HERE

Justin Fletcher Biography text here.