Recursive Hetero-Associative Memories for Translation

Mikel L. Forcada and Ramón P. Ñeco

Departament de Llenguatges i Sistemes Informàtics,
Universitat d'Alacant,
E-03071 Alacant, Spain.
E-mail: {mlf,neco}@dlsi.ua.es

Abstract. This paper presents a modification of Pollack's RAAM (Recursive Auto-Associative Memory), called a Recursive Hetero-Associative Memory (RHAM), and shows that it is capable of learning simple translation tasks, by building a state-space representation of each input string and unfolding it to obtain the corresponding output string. RHAM-based translators are computationally more powerful and easier to train than their corresponding double-RAAM counterparts in the literature.

1 Introduction

Recently, Pollack (1990) introduced a new connectionist model called the recursive auto-associative memory (RAAM). This model is capable of obtaining compact representations for compositional structures such as trees and lists, *i.e.*, is capable of representing variable-sized recursive data structures. This architecture can indeed be viewed simply as a distributed memory for storing compositional data structures. In particular, it was shown that properly trained RAAM are capable of building representations for strings acting much in the same way as a stack (strings are pushed and may then be popped).

More recently, Chalmers (1990) and Chrisman (1991) used the RAAM formalism to learn simple syntactic transformations and translations. They show how RAAM learn to perform computations directly with the distributed representations obtained by them, without accessing their compositional structure (*holistic* computations). Chrisman (1991) used a (double, stack-like) RAAM architecture in the domain of natural language translation (a small English ↔ Spanish translation task). Chrisman distinguishes two possible ways to perform the translations: *transformational* and *confluent*. In the transformational translations, the auto-association of the sentence and its translation are used to learn separate representations *before* learning the translation function between these representations. In the confluent approach, the network is trained to obtain the *same* internal representation for the sentence and its translation, *i.e.*, the original and final sentences should have identical internal representations, and they may have two different decodings, one corresponding to the original sentence, and the other to its translation. This implies that the network must be trained to auto-associate the two sentences by learning two different encoding-decoding mechanisms.

The translations performed by these models have some limitations. In particular, the tasks used by Chrisman (1991) are limited to one-to-one translations; that is, no two different sentences in one language can have the same translation in the other language. Chalmers' (1991) translators may learn many-to-one translations, but the fact that each input sentence has a unique representation forces the translation function to perform the many-to-one mapping on its own.

We propose a new model, called the RHAM (Recursive Hetero-Associative Memory), which has a computational power which is equivalent or superior to that of the RAAMs used by Chalmers (1990) and Chrisman (1991), and may be applied to learn general translations from examples in which different sentences may have the same translation. The main difference between the RAAM models used so far for translation and the new model is that in the new learning algorithm the network does not need to learn to auto-associate the input and output strings separately; we only train the network to obtain a suitable internal representation of the input, and to decode this representation to obtain the corresponding output string.

2 Definitions

We will study input strings over an input alphabet Σ; the set of all finite-length strings over Σ will be denoted Σ^*. The output strings (translations) are strings over an output alphabet Δ, that is, strings in Δ^*. The class of translations we will consider are those that may be represented by a function (an application)

$$\tau : \Sigma^* \to \Delta^*, \tag{1}$$

so that there is a single translation for each string in Σ^* but two different strings in Σ^* may have the same translation.

In this paper we consider translations that may be performed by *Mealy machines* and more general *deterministic generalized sequential machines* (DGSM) (Hopcroft and Ullman 1979, Salomaa 1987). A Mealy machine is a six-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_1)$, where Q is a finite set of states, Σ a finite set of input symbols (*input alphabet*), Δ a finite set of output symbols (*output alphabet*), $\delta : Q \times \Sigma \to Q$ the next-state function, $\lambda : Q \times \Sigma \to \Delta$ the output function, and $q_1 \in Q$ the initial state. The class of translations that a Mealy machine may perform is limited: for example, input strings and their translations always have the same length (Salomaa (1987) calls these translations length-preserving). A deterministic generalized sequential machine may be formulated as a Mealy machine in which the next-state function is $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to Q$, and the output function is $\lambda : Q \times (\Sigma \cup \{\epsilon\}) \to \Delta \cup \{\epsilon\}$, where ϵ represents the empty string. In this machine, states can have transitions defined either on the empty string or on input symbols but not on both; in addition, for the machine to halt always, it is required that no state be reachable from itself using only transitions with ϵ as input.

3 The RHAM Architecture

A RHAM, as a RAAM, consists basically of two feedforward neural networks, which we will call the *encoder* and the *decoder*. The original RAAM model can encode arbitrary tree structures, but we only need the sequential or "stack" version of Pollack's (1990) RAAM: that is, strings are encoded symbol by symbol. However, the experiments shown in this paper could be generalized to perform translations using the general RAAM version between trees (in this case we would have the restriction that the number of units used to represent a terminal symbol must be equal to the number of hidden neurons used to represent the tree). In the case of the sequential version, where we are basically encoding structures such as lists and stacks, this restriction is not necessary.

In our model each input string $w \in \Sigma^*$ has a vector representation in $[0,1]^N$ where N is the number of hidden units in the hidden layer. A function

$$\mathbf{r} : \Sigma^* \to [0,1]^N \tag{2}$$

assigns a (not necessarily distinct) vector representation to each string.

Let $\mathbf{r}_0$ be the representation of the empty string ϵ or "nil":

$$\mathbf{r}(\epsilon) = \mathbf{r}_0; \tag{3}$$

then, the representation of any string wa ($w \in \Sigma^*$, $a \in \Sigma$) is

$$\mathbf{r}(wa) = \text{encode}\left(\mathbf{r}(w), \mathbf{u}_a\right) \tag{4}$$

where $\mathbf{u}_a$ is a vector representing symbol a, and

$$\text{encode} : [0,1]^N \times [0,1]^K \to [0,1]^N, \tag{5}$$

is the encoding function which has a simple connectionist implementation; $\mathbf{r}[t] = \text{encode}(\mathbf{r}[t-1], \mathbf{u}[t])$ is realized as a simple perceptron:

$$r_i[t] = g\left(\sum_{j=1}^{N} W_{ij}^{rr} r_j[t-1] + \sum_{k=1}^{K} W_{ik}^{ru} u_k[t] + W_i^r\right) \tag{6}$$

where the W's represent weights and biases and $g(x) = 1/(1+\exp(-x))$; that is, a single-layer feedforward neural network having $N+K$ input units and N output units, and therefore $(N+K+1)N$ different parameters.

With the recursive function $\mathbf{r}$, a string w of length L_w can be encoded by placing a representation $\mathbf{u}[1]$ for the first symbol of the string in the left-hand K units of the input, and a representation for the empty string, $\mathbf{r}(\epsilon) = \mathbf{r}[0]$, on the right-hand N units of the input. Then, the encoder produces a internal representation for a string consisting of the first symbol on the N hidden units. The activations of this hidden units are then copied recursively to the rightmost N units of the input and the representation $\mathbf{u}[2]$ of the second symbol of the string is placed on the left-hand K units; the process is repeated until all the symbols of the string are processed and a representation $\mathbf{r}[L_w] = \mathbf{r}(w)$ for the whole string is obtained.

The decoder, when properly trained, unfolds the representations of input strings to obtain the corresponding output strings. States in $[0,1]^N$ are also interpreted as representations of output strings; and the input representation $\mathbf{r}(w)$ of string w is taken to be the output representation $\mathbf{s}(x)$ of its translation $x = \tau(w)$.

The output representation function $\mathbf{s}$ is analogous to the input representation function $\mathbf{r}$:

$$\mathbf{s} : \Delta^* \to [0,1]^N. \tag{7}$$

The decoder works as follows: the representation $\mathbf{y}_b$ of a symbol b from the output alphabet Δ is popped from the representation $\mathbf{s}(xb)$ of string xb and the representation $\mathbf{s}(x)$ of string x is simultaneously produced. Popping of a complete string ends when a special representation $\mathbf{s}_0$, representing the empty string over the output alphabet, appears in the hidden layer.

The implementation of the decoder is also a perceptron with N input units and $N+M$ output units, having therefore $(N+M)(N+1)$ different parameters (weights and biases). The first N outputs correspond to the representation of the rest of the output:

$$s_j[t-1] = g\left(\sum_{i=1}^{N} W_{ji}^{ss} s_i[t] + W_j^s\right), \tag{8}$$

and the remaining M outputs to the representation y of the popped output symbol

$$y_k[t] = g\left(\sum_{i=1}^{N} W_{ki}^{ys} s_i[t] + W_j^y\right). \tag{9}$$

If we have the internal representation of an input string encoded in the hidden units, then we can obtain the translation by repeating the decoding process until the end of the string is detected (the representation $\mathbf{s}_0$). In the first iteration, the leftmost M units of the output return the local representation $\mathbf{y}[L_x]$ for the last symbol of the string x, while the rightmost N units return the representation for the remainder of the string.

The representation $\mathbf{y}_b \in [0,1]^M$ of each output symbol b and the representation of the empty output string $\mathbf{s}_0 \in [0,1]$ may be chosen in advance but might also be learned during the learning. The representation of the empty input string $\mathbf{r}[0] \in [0,1]^N$ may also be chosen or learned.

Figure 1 shows schematically the architecture of a RHAM. In the example the encoder reads the input "1011", symbol by symbol, to obtain a representation of the input sentence $\mathbf{r}$("1011"). This representation is also the representation of the translation of this sentence, $\mathbf{s}$("001"). The representations of the symbols of the translated sentence are popped by the decoder: first symbol '1', second, symbol '0', and then, symbol '0' and in the right side of the decoder's output, the

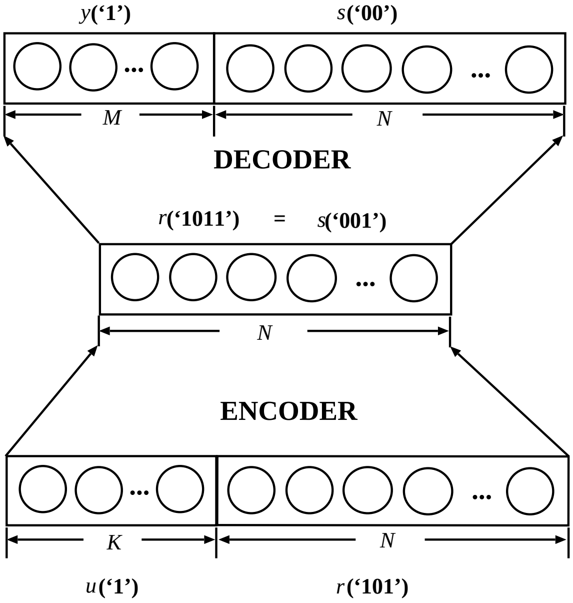

Fig. 1. Architecture of a RHAM performing one step of the translation of sentence "1011" into sentence "001".

representation for the empty output string ($\mathbf{s}_0$), indicating that the translation has finished. Note that if $\tau(x) = x$, $\forall x$, $\Delta = \Sigma$ and $\mathbf{r}_0 = \mathbf{s}_0$, this is Pollack's (1990) RAAM acting as a stack.

4 Training

4.1 Error Function

We train the network by minimizing an error function for a training sample S containing translations t from $\Sigma^* \times \Delta^*$ which are compatible with the existence of a single-valued function $\tau : \Sigma^* \to \Delta^*$. This error function reflects the difference between the output obtained by the net and the desired output. The total error E for the sample is

$$E = \sum_{t \in S} e_t \tag{10}$$

where e_t is the error for each translation t. In the rest of this section, we will drop the subscript t to alleviate the notation. Assume that the sample translation is (w, z). First, the string w is "pushed" into the RHAM by recursive application of the encoder, to obtain a representation $\mathbf{r}(w)$ which is taken to be the representation $\mathbf{s}(z)$ of its translation $z = b_1 b_2 \dots b_{|z|}$. The total error in the translation is the departure of the successive output symbol vectors $\mathbf{y}[|z|], \mathbf{y}[|z| - 1], \dots, \mathbf{y}[1]$

from their desired values $\mathbf{y}_{b_{|z|}}, \mathbf{y}_{b_{|z|-1}}, \ldots, \mathbf{y}_{b_1}$ plus the departure of the final representation $\mathbf{s}[0]$ from that of the empty output string $\mathbf{s}_0$:

$$e = \frac{1}{2}\left[\sum_{l=|z|}^{1} \|\mathbf{y}[l] - \mathbf{y}_{z_l}\|^2 + \|\mathbf{s}[0] - \mathbf{s}_0\|^2\right] \tag{11}$$

The values of the local representations of the output symbols $\mathbf{y}_b$ and the empty output string $\mathbf{s}_0$ may be chosen in advance (as in this paper) or allowed to change during learning. In the latter case, they might be taken to be the instantaneous average (over the whole sample) of the actual values observed where they should have appeared, but in that case, an additional penalty term would be needed to keep the $\mathbf{y}_b$'s as far apart from each other as possible, in order to avoid the collapse of all of them into a single value. Details of this procedure will be reported elsewhere.

4.2 Training Algorithm

The above formulation allows for the use of gradient-descent-based methods, such as a modified version of RTRL (real-time recurrent learning, Williams and Zipser 1989) or an adaptation of backpropagation through time (Rumelhart *et al.* 1986). This approach to training RHAMs will be reported elsewhere. In the preliminary experiments reported here we have chosen a non-gradient method, Alopex (Unnikrishnan and Venugopal 1994), which is related to simulated annealing but instead of being driven by changes in the error function it is driven by correlations between changes in error and changes in each particular weight. The method relies only on successive evaluations of the error function and needs no information about the particular structure of the system being trained. In a related task (Ñeco and Forcada 1996) where discrete-time recurrent neural networks were trained to perform simple formal translations similar to those shown here, the method proved to be a very attractive alternative to gradient descent, specially when the architecture is new (as in Ñeco and Forcada 1996) or being used in a new way for the first time (as here).

5 Experiments

We have made experiments in order to explore the translation capability of this new architecture. This experiments include simple translation tasks performed by Mealy and deterministic generalized sequential machines.

We train the network with the translations performed by the three automata shown in figure 2. These automata perform translations between strings over the alphabets $\Sigma = \Delta = \{0, 1\}$. We trained networks with $N = 3, \ldots, 12$ hidden neurons and encoded both input and output symbols with the unary (one-hot) encoding, using Alopex (Unnikrishnan and Venugopal 1994) with a step-size $\delta = 0.01$, and an initial temperature $T = 1000$. The training set contained the translations of all strings from length 1 to 8, and we check the generalization for the whole set of strings from length 9 to 10. We stopped training the net when it correctly translated all the sentences in the learning set. A successful translation is reported when all the values of the output neurons depart less than a tolerance $\xi = 0.2$ from their desired values. We consider that a network has failed to learn the task when it takes more than 500,000 epochs.

Table 1 shows the results for the first automaton, which is a Mealy machine. In this table we present the number of epochs needed to learn the task and the generalization performance of the net, for $N = 2, \ldots, 12$ neurons. With bigger values of N, we observed that the net either fails to learn the translation task, or has worse generalization performance. The best generalization result was obtained for $N = 6$ hidden neurons, with 156,000 epochs of Alopex.

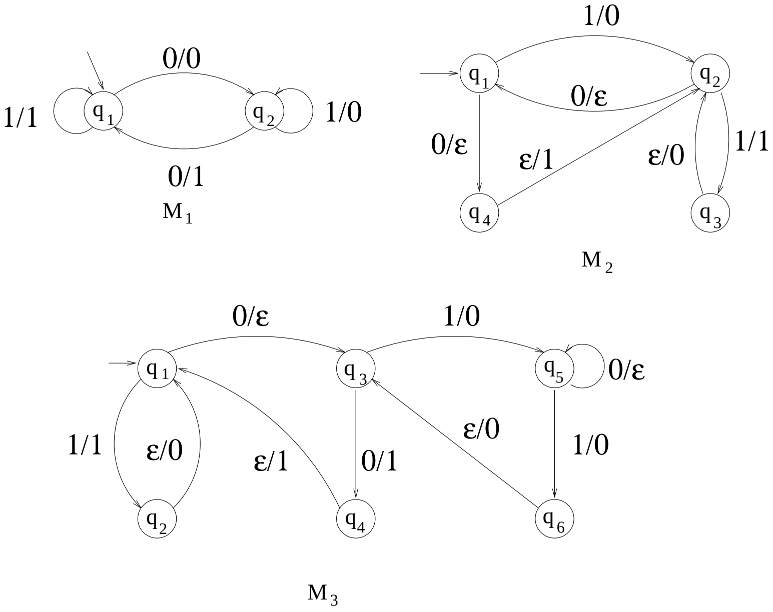

Fig. 2. Automata used in the experiments

In the second experiment we used a DGSM (automaton M_2 shown in figure 2). Note that this automaton performs a many-to-one translation in the sense that two different input strings may have the same output string as translation: for example, $\tau(\text{“0”}) = \tau(\text{“00”}) = \text{“1”}$. The results obtained for this task are shown in table 2. In this case we obtained the best generalization result (87%) for $N = 9$ neurons, with 158,000 epochs, but for $N > 9$ we observed that the generalization results are worse for this task (60 % for $N = 10$, 55 % for $N = 11$, 63% for $N = 12$).

In the third experiment we used automaton M_3 shown in figure 2. This automaton also performs a many-to-one translation. For example, $\tau(\text{“10”}) = \tau(\text{“1”})$

Table 1. Results for automaton M_1. We show the number of Alopex epochs (in thousands) used to learn the training set, and the generalization results (percentages) for all input strings from length 9 to 10.

Neurons	kepochs	Generalization
3	failed, failed, 254, failed	–, –, 65%, –
4	215, 250, failed, failed	68%, 62%, –, –
5	115, 105, 98, 117	67%, 77%, 60%, 78%
6	160, 177, 156, 189	90%, 85%, 91%, 82%
7	98, 126, 117, 108	85%, 87%, 82%, 90%
8	177, 155, 165, 140	75%, 73%, 81%, 79%
9	77, 90, 86, 105	75%, 74%, 55%, 76%
10	155, 284, failed, failed	30%, 47%, –, –
11	failed, 430, 350, failed	–, 53%, 42%, –
12	442, 270, failed, failed	59%, 46%, –, –

Table 2. Results for automaton M_2. We show the number of Alopex epochs (in thousands) used to learn the training set, and the generalization results (percentages) for all input strings from length 9 to 10.

Neurons	kepochs	Generalization
3	failed, 350, failed, failed	–, 53%, –, –
4	270, failed, 362, 388	60%, –, 42%, 31%
5	226, 270, 231, 273	70%, 68%, 55%, 80%
6	210, 224, 230, 205	74%, 73%, 52%, 75%
7	175, 170, 215, 265	70%, 75%, 80%, 65%
8	230, 180, 221, 192	70%, 76%, 71%, 75%
9	150, 124, 290, 158	67%, 84%, 82%, 87%
10	253, 306, 379, 366	45%, 50%, 60%, 48%
11	failed, 340, 365, failed	–, 45%, 55%, –
12	220, 166, 252, 330	50%, 55%, 63%, 54%

= “10”. The results obtained for this automaton are shown in table 3. In this case, we obtain the best generalization result (89%) for $N = 8$ neurons. In this experiment, we consider that a network has failed when it takes more than 800,000 epochs.

6 Discussion and Conclusions

The results obtained for examples of the widest class of deterministic finite-state translations (deterministic generalized sequential maps, Salomaa 1987) show that recursive hetero-associative memories (RHAM) are a promising approach to the neural induction of translators from examples. The RHAM architecture has

Table 3. Results for automaton M_3. We show the number of Alopex epochs (in thousands) used to learn the training set, and the generalization results (percentages) for all input strings from length 9 to 10.

Neurons	kepochs	Generalization
3	failed, failed, failed, failed	–, –, –, –
4	failed, failed, failed, failed	–, –, –, –
5	450, failed, 572, failed	78%, –, 65%, –
6	failed, 550, 309, 674	–, 45%, 63%, 75%
7	420, 580, 331, 427	65%, 60%, 53%, 63%
8	failed, 494, 537, failed	–, 75%, 89%, –
9	610, 537, 540, 593	85%, 72%, 87%, 63%
10	failed, 510, 638, failed	–, 70%, 35%, –
11	failed, failed, 705, failed	–, –, 62%, –
12	failed, failed, failed, failed	–, –, –, –

been shown to be computationally superior (they may deal with many-to-one translations) to RAAMS as used by Chrisman (1991) and theoretically equivalent to RAAMS as used by Chalmers (1990), but architecturally simpler and easier to train (there is no need to train the network to auto-associate input and output strings separately to learn the translation tasks).

Generalization results are good if we take into account that they refer to strings that are always longer than those in the training set[1]. We are currently studying a new training strategy which forces representations of *all* prefixes of *all* strings in the training set to be *decodable* into prefixes of output strings; we believe that this will improve the generalization performance as well as yield partial translations as a byproduct. The results of this approach and a detailed study of the representations learned by the network will be reported elsewhere.

The approach shown here is complementary to the one involving a new recurrent neural network architecture presented by us (Ñeco and Forcada 1996), which has also been trained to learn deterministic generalized sequential machines. We are currently studying the relationships between both approaches, with a possible hybrid approach in mind.

Acknowledgments: This work has been supported through grant TIC95-0984-C02-01 of the Spanish Comisión Interministerial de Ciencia y Tecnología (CICyT). Ramón P. Ñeco is supported by the Generalitat Valenciana (Spain).

References

Chalmers, D.J. (1990) "Syntactic Transformations on Distributed Representations", *Connection Science* **2**, 53–62.

[1] Chrisman (1991) and Chalmers (1990) always tested generalization on strings in the same length range.

Chrisman, L. (1991) "Learning Recursive Distributed Representations for Holistic Computation", *Connection Science* **3**:4, 345–366.

Hopcroft, J.E., Ullman, J.D. (1979) *Introduction to automata theory, languages and computation.* Reading, Massachussets: Addison-Wesley.

Ñeco, R., Forcada, M.L. (1996) "Beyond Mealy machines: Learning translators with recurrent neural networks", *Proc. World Congress on Neural Networks* (San Diego, Calif., Sept. 1996), p. 408–411.

Pollack, J.B. (1990) "Recursive distributed representations", *Artificial Intelligence* **46**, 77–105.

Rumelhart, D.E., Hinton, G.E., Williams, R.J. (1986) "Learning internal representations by error propagation". In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (D.E. Rumelhart and J.L. McClelland, eds.), Vol. 1, Chapter 8, Cambridge, MA: MIT Press.

Salomaa, A. (1987) *Formal Languages*, Boston, Massachusetts: Academic Press.

Unnikrishnan, K.P., Venugopal, K.P. (1994) "Alopex: A Correlation-Based Algorithm for Feedforward and Recurrent Neural Networks", *Neural Computation* **6**, 469–490.

Williams, R.J., Zipser, D. (1989) "A learning algorithm for continually running fully recurrent neural networks", *Neural Comp.* **1**, 270–280.

This article was processed using the LaTeX macro package with LLNCS style