Tingyu Thomas Lin
Juan M. Navarro
CS 263B – Natural Language Processing
Final Project Report
Due: Monday, June 8th, 2009

A Perceptron-Based Approach for Image-Word Association

1. Project Purpose

Our project had 2 goals:

1. ***Associate an image to a concrete noun:*** The goal is to associate concrete nouns with an image. For example, if we feed the word "tree" into our system, we want the system to generate an image of a tree.
2. ***Understand relative positioning of objects:*** Our ultimate goal for our system is to have our system recognize relative positions of objects. For example, if we feed "house right car", our system should generate an image of a house to the right of a car.

We separated our system into two modules. The first module, which we call the *image module*, accomplished the first goal. The second module, which we call the *relations module*, accomplishes the second goal, and it also uses the first module as a subcomponent. The motivation behind this split is that a human can learn images independent of understanding relative positions of objects. Also, a human in all likelihood needs to know a few images before learning the concepts of left and right, but once a human learns left and right the human should be able to learn how a new object looks like and place it next to another known object without relearning left and right. This ability is equivalent to us replacing the image module with another image module that knows more objects.

2. Image Module

2.1 Input/Output Specifications

The input to this module is a concrete noun encoded in 7-bit ASCII. We fixed the character length of the noun to 8, for a total of 56 bits. If the noun is not eight characters long, we appended them with spaces. The output is a 10x10 grayscale image of the associated concrete noun. The pixels have values of 0 to 1, where 0 is black and 1 is white.

We created by hand input/output pairs for this module to learn.

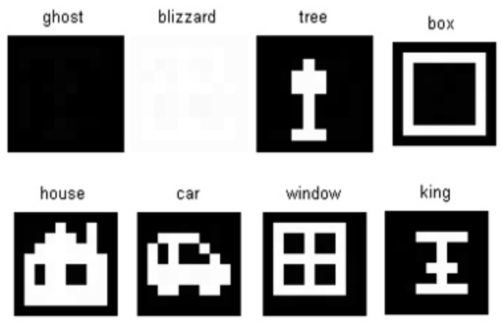

Figure 1 Sample input/output pairs.

2.2. Architecture

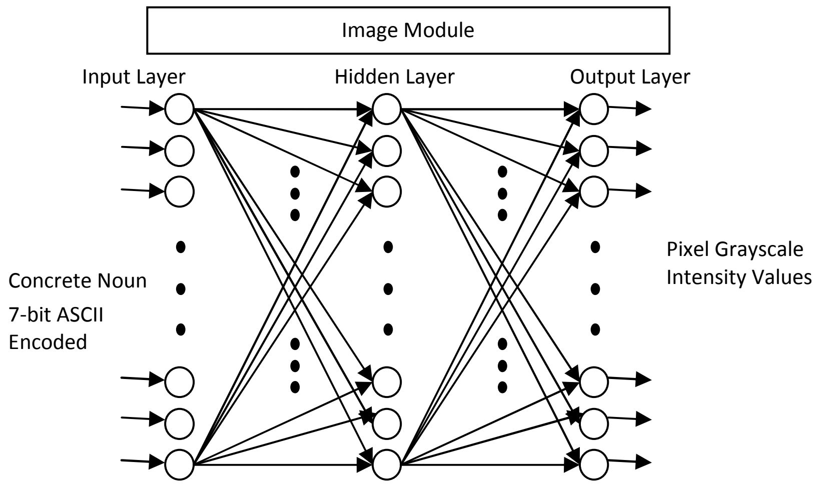

Figure 2. Overview of the Image Module's architecture.

We use a three-layer perceptron for this module:

Input Layer

We had 56 input layer nodes, one for each bit of the input noun. The nodes outputted the value of the bit it represented.

$$I_i^n = the\ ith\ bit\ of\ noun\ n's\ input$$

Hidden Layer

We applied the logistic sigmoid function $\sigma(t) = \frac{1}{1-e^t}$ at the outputs of the hidden layer to get nonlinear combination of layers and squash outputs to between 0 and 1. The latest version of our code has 7 hidden neurons. We played with the size of the hidden layer and going below seven neurons compromised the perceptron's ability to learn and produce desired outputs. If X_{ji} is the weight between the ith input node and the jth hidden node, then the output of the jth hidden node when the nth noun is fed as input is:

$$H_j^n = \sum_i \sigma(X_{ji} I_i^n)$$

Output Layer

We had 100 output nodes, one for each pixel of the image associated with input noun. Like the hidden layer, we used the logistic sigmoid function at the outputs of this layer. If Y_{kj} is the weight between the jth hidden layer and the kth output node, then the output of the kth output node for the nth input noun is:

$$O_k^n = \sum_j \sigma(Y_{kj} H_j^n)$$

2.3. Learning

We tried two learning algorithms to train our system:

Gradient Descent

For our first attempt, we used online gradient descent to train the network. The error function we defined for gradient descent was:

$$E^n = \frac{1}{2} \sum_k (T_k^n - O_k^n)^2$$

where T_k^n is the target output for the kth pixel of the nth image, and O_k^n is the actual output for the kth pixel when the nth image was fed as input. The kth pixel can be thought of as the index of the 2D image if it was reshaped into 1D. The pseudocode then for gradient descent is:

```
Randomize $X_{ji}$, the weights of connections between the $i$th input and the $j$th hidden nodes
Randomize $Y_{kj}$, the weights of connections between the $j$th hidden and the $k$th output nodes
While the average error across all images $n$ is greater than a small value:
    For each image $n$:
        Calculate all $\frac{dE^n}{dY_{kj}}$ and $\frac{dE^n}{dX_{ji}}$ for the specified $n$ via the following equations:
            $\delta O_k^n = -O_k^n(1 - O_k^n)(T_k^n - O_k^n)$
            $\delta H_j^n = -H_j^n(1 - H_j^n)\sum_k Y_{kj}\delta O_k^n$
            $\frac{dE^n}{dY_{kj}} = \delta O_k^n H_j$
            $\frac{dE^n}{dX_{ji}} = \delta H_j^n I_i$
        Apply weight updates with:
            $Y_{kj} = Y_{kj} - \lambda \frac{dE^n}{dY_{kj}}$
            $X_{ji} = X_{ji} - \lambda \frac{dE^n}{dX_{ji}}$
```

We set the "small value" at 10^{-7}. Though this method learned our eight image training data set and terminated on the small error condition, we did not like online gradient descent for two reasons:

Learning rate: We had little intuition to setting a good learning rate beyond simply adjusting the weight. For a four-image training set, we found a learning rate of 1 yielded pretty fast convergence, but when we moved up to a seven-image set, our system refused to converge until we dropped our learning rate to .8. It is highly impractical to manually adjust the learning rate when we increase the data set size.

Slow convergence: With the learning rate set at .8 and a seven-image set, our system converged in 1737 epochs (see Figure C.1 in Appendix). Looking at plots of the error as the system learned our images, we noticed the error would rapidly drop over the first couple hundred epochs but then slow to a crawl over the last 1000+ epochs when the error gets close to zero. This slowdown is a direct result of the partial derivates $\frac{dE^n}{dY_{kj}}$ and $\frac{dE^n}{dX_{ji}}$ falling close to zero as the overall error also falls towards zero; gradient descent is updating the weights with progressively smaller and smaller steps.

Rprop
Rprop is a batch learning variant of gradient descent; instead of updating the weights after each image in the training set, it sums the partial derivatives across all the images. Rprop also only considers the direction of the partial derivatives by using a separate automatically adjusting weight step for modifying weights, though weight steps are still applied in the same direction as the partial derivative. If the direction of the partial derivative remains unchanged from one iteration to the next, i.e. the sign of the partial remains the same, the weight step is increased. If the direction of the partial derivative changes, then the weight step is reduced, as a change in sign indicates that the previous weight step has overstepped the minimum. These modifications directly alleviate our two issues with gradient descent. We implemented Rprop by following the original paper that introduced this algorithm [4].

We again quit Rprop after the average error across all images fell below the same 10^{-7} threshold. For a seven-image training set, Rprop converged in 160 epochs, a drastic improvement over the 1737 epochs that gradient descent took (see Figure D.1 in Appendix).

3. Relations Module

3.1. Input/Output Specifications

The input to the relations module takes the specific form:

<noun1> <relation> <noun2>

Both <noun1> and <noun2> must be items learned by the image module. The encoding also remains unchanged, so the two nouns are still 56 bits wide. <relation> can be one of the four relationships:

1. right
2. left
3. above
4. below

We fixed the width of the relations to 5 characters wide, with spaces padding the right of the word if it is not 5 characters wide, and encoded the word in 7-bit ASCII. We also place spaces between <noun1> and <relation>, and again between <relation> and <noun2>, so the relation word was 7 characters wide, or 49 bits wide. This is a total of 161 bits to represent the entire input phrase.

As for output, we created 20x20 grayscale images. Since there were 8 images our image module learned and there were 4 possible relations, we had a total of 256 input/output pairs (see Figure E.1 in Appendix).

3.2. Train vs. Test Set

Of the 256 possible input/output pairs, we randomly selected 16 images and called that set the test set. The remaining 240 phrase/image pairs we used as the training set. The hope was that after training on the 240 images, we could feed the 16 test images and obtain an image with objects placed correctly next to each other, despite never having explicitly training on those images.

3.3. Architecture

We used a three-layer perception again, but we added an extra layer before the input layer:

Pre-input layer

This layer took the two nouns and ran them through the image module. It then fed the outputs of the image module and the relation word into a three-layer perceptron. This layer is what gives the relations module the knowledge of what concrete nouns look like. We hard coded the connections between the input and the image modules.

3-Layer perceptron

The input layer had 249 nodes, as two images and a relation word combines to 249 bits. The output layer had 400 nodes, one node for each pixel in the 20x20 output image. The hidden layer was set to 50 nodes. We tried sizes of 30, 50, 100, and 200 for the hidden layer and 50 seemed to do the best, though we did not do a quantitative analysis. We used the same logistic sigmoid function as we did in the image module.

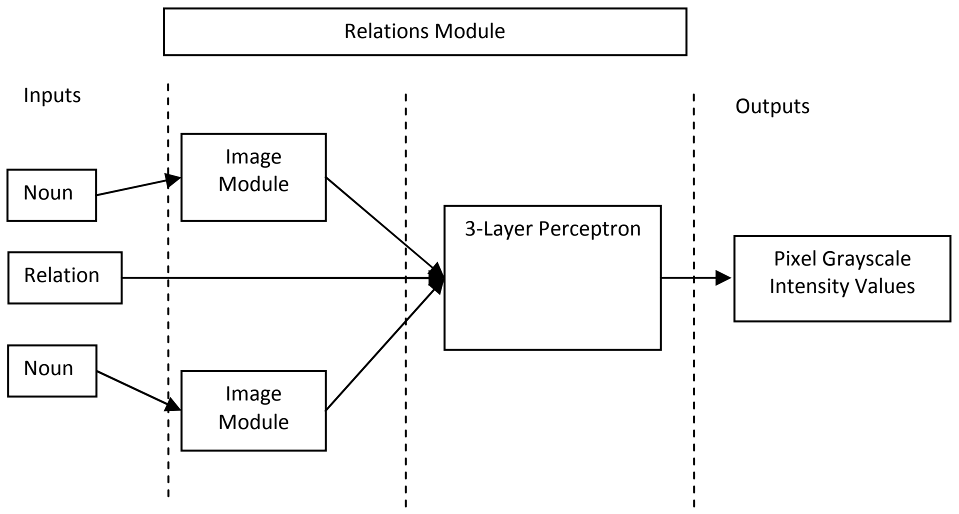

Figure 3. Overview of the Relations Module's architecture.

3.4. Learning

We attempted four different learning schemes: gradient descent, Rprop, iRprop+, and iRprop+ with a different cost function:

Gradient Descent
The failure for gradient descent in training our relations module is what spurred us into implementing Rprop in the first place. We had to set the learning rate to .003 to prevent the system from diverging on the very first epoch, but with such a small learning rate, the error fell at a snail's pace. After 20,000 epochs, which took about 4 hours on a 5-year old laptop running Matlab, we cancelled the execution. The error was hovering around .03, and the change in error after each epoch was so small that we could no longer see a change despite printing the error out to 10 decimal places.

Rprop
Rprop produced mixed results. We first tried learning a stripped down test set. We took out one of the nouns from the image learning module, which reduced the total number of relation phrase/image pairs to 7x7x4=196. We still took out 16 images randomly to create our test set, so we trained our system with now only 180 images. With this reduced set, Rprop converged in 156 epochs. However, though the training set was learned with less than an average error of 10^{-7}, the test set was not perfect (see Figure F.1 in Appendix). We suspected that increasing the test set would improve results. However, by going back to the original full 256 image set, Rprop refused to converge, instead opting to sit at a local minimum where the error oscillated at some value well above the 10^{-7} threshold we set.

iRprop+
iRprop+ is an improved version of Rprop [5]. Rprop will reduce the weightstep if the partial derivative changed directions; iRprop+ takes that one step further and will roll-back the weight to what it was in the previous epoch if the overall error increased. The rationale is that the weight that the overshot the local minima is what caused the error to increase and thus should be returned to what it was before.

This method did not improve anything. For the reduced 196 set, there was no notable increase in epochs needed to converge, and for the 256 set, iRprop+ still got stuck.

iRprop+ with a different cost function

When observing iRprop+ trying to learn the 256 image set, invariably it would set a few pixels incorrectly to 0 or 1 and find itself unable to escape that pit. We hypothesized that this was caused by iRprop+'s batching strategy. Say, for a particular image, the partial derivative $\frac{dE}{dY_{kj}}$ was positive, so the weight is being decreased. For one particular input n, $\frac{dE^n}{dY_{kj}}$ is a very large negative value, and if the weight isn't increased, the net will never converge. However, all other images produce a small yet positive partial derivative for that weight. When iRprop+ adds together all the partial derivatives, it is possible that the batched partial derivative will end up positive, if there are enough small positive partial derivatives to "outvote" the single large negative weight. This will drive the weight down until it is a very large negative value.

Now, since $O_k^n = \sum_j \sigma(Y_{kj} H_j^n)$, if the hidden layer output that crosses that incredibly negative weight is even slightly positive, the product $Y_{kj} H_j^n$ will be large and the output pixel will be instantly pulled to 1. Likewise, if the hidden layer output is negative, $Y_{kj} H_j^n$ will be very negative and pull the pixel value to 0. Once it gets to this point, iRprop+ lacks the capability to escape, as $\frac{dE^n}{dY_{kj}} = \delta O_k^n H_j$ and once the pixel value is 0 or 1, $\delta O_k^n = -O_k^n(1 - O_k^n)(T_k^n - O_k^n)$ will always be 0, and so $\frac{dE^n}{dY_{kj}} = \delta O_k^n H_j$ will also be 0.

A potential solution then is to prevent a large $\frac{dE^n}{dY_{kj}}$ value from being outvoted. To achieve this we tried using a modified cost function:

$$E^n = \frac{1}{m} \sum_k (T_k^n - O_k^n)^m$$

This changes δO_k^n to:

$$\delta O_k^n = -O_k^n(1 - O_k^n)(T_k^n - O_k^n)^{m-1}$$

If $m = 2$, the cost function is identical to the original cost function. For even values of $m > 2$, the term $(T_k^n - O_k^n)^{m-1}$ pulls pixels with small errors towards zero, since $(T_k^n - O_k^n)$ is small, making $\frac{dE^n}{dY_{kj}}$ also small. However, for pixels that are completely wrong, $(T_k^n - O_k^n)^{m-1}$ is close to 1, so $\frac{dE^n}{dY_{kj}}$ does not plummet nearly as fast. This makes images with erroneous pixels much less likely to be outvoted, giving pixels that are drastically wrong more influence over the weights.

This did not get our system to converge; however, setting m as 6 and letting the system run for 2,000 epochs, the error fell to about .0008 (see Figure G.1 in Appendix). While not quite below our 10^{-7} threshold, the error was still slowly falling, though it is unclear whether the system will eventually find itself stuck in a local minimum.

3.5. Evaluation of Relations Module

To evaluate our relations module, we fed our 16-image test set through the relations module after training the perceptron using iRprop+ with the new weight function:

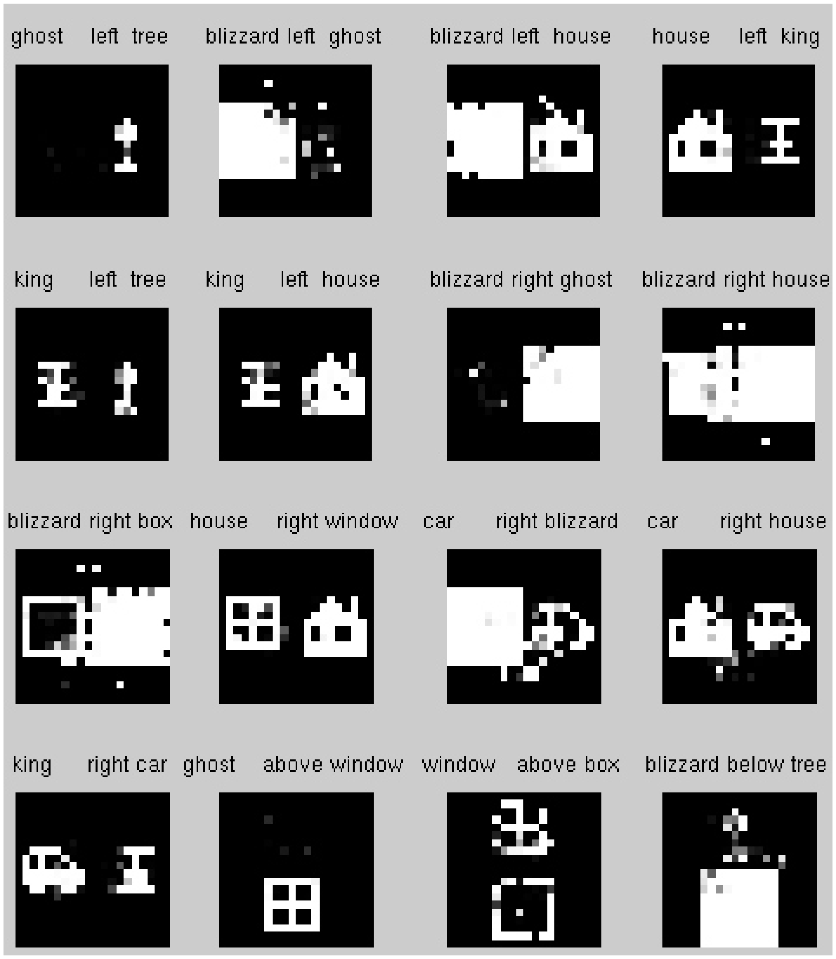

Figure 4 – Results of iRprop+ with new weight function

Our system, when learning images, never managed to get the error to fall below 10^{-7}, so we did not expect the system to produce good images for our test set. The test set had an MSE of 0.0204, which is a far cry from our system error threshold of 10^{-7}.

To further evaluate our system, we introduced the stapler, a new noun/image pair, into the image module. Since all our catastrophic forgetting avoidance implementations don't work very well, if at all, we simply just trained a new set of weights for the image module's perceptron using this new image along with the old eight images.

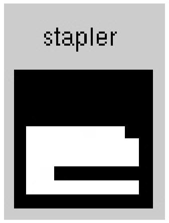

Figure 5 – Additional noun/image pair for the image module.

Since there are now 9 objects that can be to the right, to the left, above, and below of each other, we now have a total of 324 possible relationship pairings, of which 64 are new and have the stapler in them. We ran this new set of 64 through our system to test if it had indeed learned the concepts of right, left, above, and below independent of the images. The following is sixteen images randomly selected out of the 64 images involving a stapler:

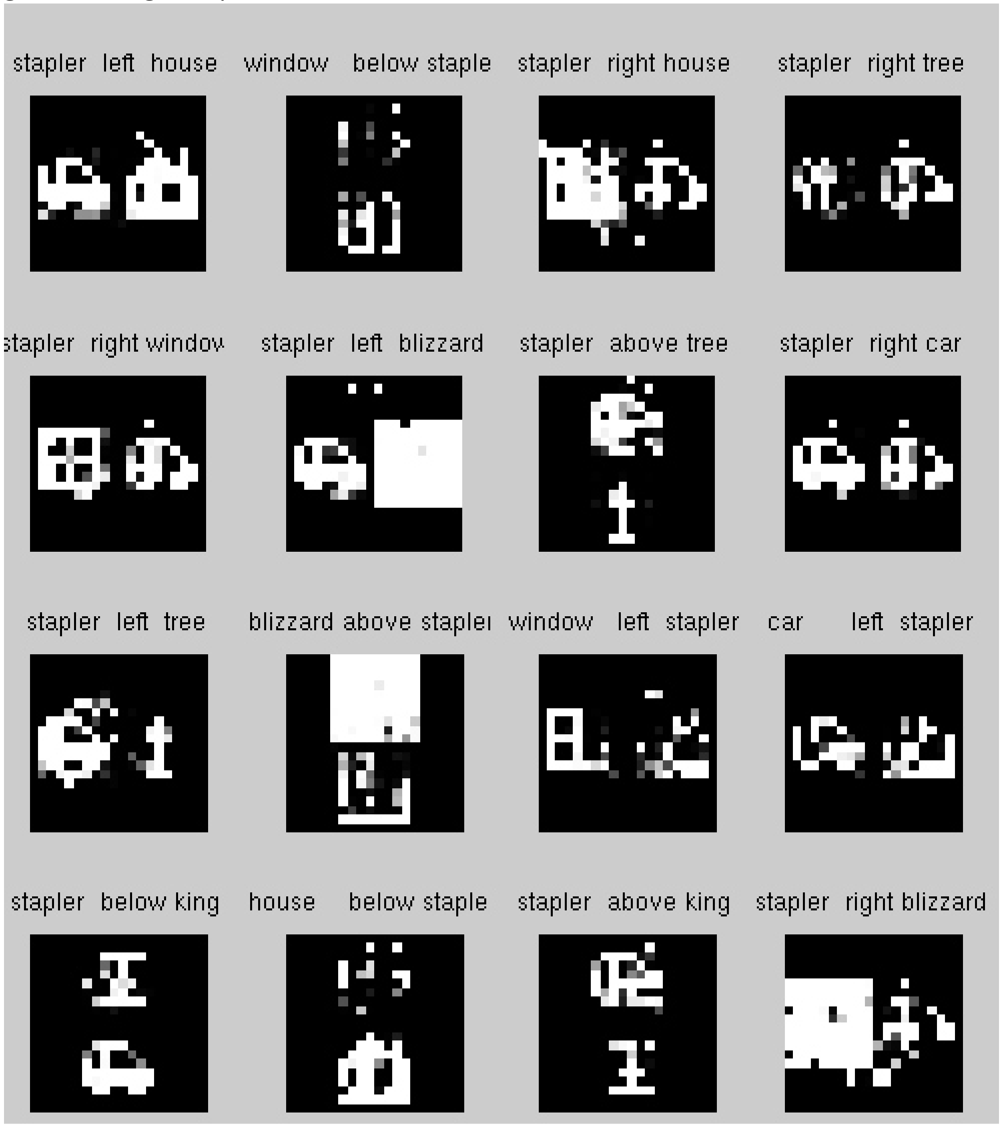

Figure 6 – Results after adding stapler input

The MSE across the 64 images was .2386. A uniformly gray image, where all output nodes produce .5 for their output, has an MSE of .25, so we can confidently say our relations module has not separated the concepts of objects and relations apart.

4. Current Status

Beyond attempting to refine the relations module to produce better results, we were attempting to extend the image learning module to learn images without catastrophic forgetting. We explicitly rehearsed all previously known noun/image pairs to learn new images; however, we wanted to mimic a human's ability to learn how things look like without explicitly studying past inputs.

We tried two approaches and considered a third:

4.1. Simple Pseudo-patterns

Research [2] presented the idea of simple pseudo-patterns: a neural net is bombarded with noise to generate associated output vectors. Each input-output pair collected is a reflection of the function previously learned in the network. We use 15 static pseudo-patterns in our implementation; the results of this approach are shown in Appendix B. Figure B.1 shows the original set of word/image pairs learned by the net. Figure B.2 shows the learned pseudo-pattern sequence along with the new word/image association to learn (i.e. diagonal). The results, shown in Figure B.3, seem promising: the network is able to vaguely remember the associations for blizzard, tree, house, window, and box, albeit with some noise.

4.2. Two coupled neural networks

Two neural nets, NET1 and NET2, are used to help prevent catastrophic forgetting. After NET1 has learned the original set of word/image associations, it is bombarded by a noise generator. The neural activity (inputs/outputs) of NET1 are captured and learned by NET2. After NET2 has learned the pseudo-patterns from NET1, NET1 is then given an external input to learn. In our implementation, this external input is the diagonal shown in Figure A.4; the noise consists of a randomly generated 7-bit ASCII string 8 characters in length. While NET1 learns the new input, it is also trained with pseudo-patterns generated by NET2 after NET2 has been bombarded with noise. Figures 7 and 8 below illustrate this process:

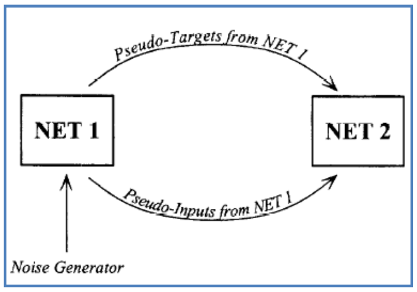

Figure 7 – Noise is fed into NET1 to generate pseudo-patterns for NET2

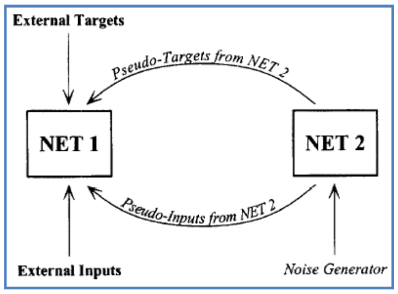

Figure 8 – Noise is fed into NET2 to generate pseudo-patterns for NET1

We use 7 static pseudo-patterns in our implementation; the results of this approach are shown in Appendix A. Figure A.1 shows the original images learned by NET1. After NET1 has learned the associations in Figure A.1, it is then bombarded with noise, and the results shown in Figure A.2. These associations (inputs and outputs) are then transferred to NET2 for learning. Noise is then fed to NET2, and the generated pseudo-patterns, along with external input, are given to NET1 for learning (Figure A.4). Finally, NET1 is tested with the original inputs to see if catastrophic forgetting has occurred. Our results in Figure A.5 shows that with our current architecture, catastrophic forgetting is not prevented.

Our research [1] indicated that catastrophic forgetting can be prevented by using a reverberating network structure, as shown in Figure 9:

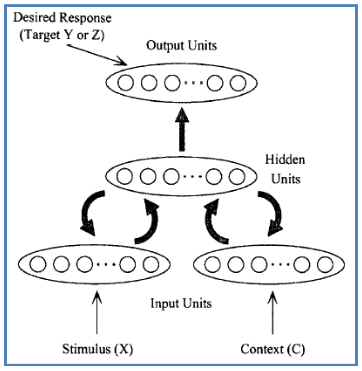

Figure 9 – Architecture of a reverberating network

In such a network, the deep structure of the distributed information represented in the connection weights is captured by feedback provided to the input layer from the hidden layer. The architecture described in [2] implies that noise is generated after every iteration (i.e. epoch); Juan's implementation generates a static noise vector for all epochs.

4.3 Two coupled neural networks with reverberation

Juan's initial impression was that he could use Elman networks to implement catastrophic forgetting since the reverberating network in [2] uses an architecture similar to an Elman network. Unfortunately, we had not realized that the architecture described in [1] to help prevent catastrophic forgetting was a different architecture than the Elman [3] network. Hence, our tests to help prevent catastrophic forgetting with two reverberating coupled neural nets were not successful.

Work Division

Task	Juan	Thomas
Implementation		
• Image Module		
○ Creating Input/Output Data	50%	50%
○ Perceptron Architecture Implementation	50%	50%
▪ Gradient descent	50%	50%
▪ Rprop	—	100%
○ Catastrophic Forgetting Avoidance	100%	—
• Relations Module		
○ Creating Input/Output Data	—	100%
▪ iRprop+, iRprop+ w/ alternate error function	—	100%
Presentation and Paper	50%	50%