Learning Rules from Understandable Weight Matrices of Feedforward Neural Networks Using Competitive Distribution of Error Signals

Matthew J. Radio
University of Maryland
Department of Computer Science
College Park, MD 20742
radio@cs.umd.edu

Abstract

Neural networks are a widely used method for classification tasks and function approximation. Although, the technique is very powerful, the representation of the trained network leaves a lot to the imagination when it comes to understanding how the network learned the particular function. This paper introduces a new method for trainining a feedforward neural network so that the resulting weight matrix is more understandable to the human eye. The method employed utilizes competitive activation dynamics along with a competitive distribution of the error signals in the backward mechanism. The method was found to be useful in a very simple problem for extracting rules.

1 Introduction

Pattern classification is one of the most common application of neural networks. While neural networks are extremely powerful in the scope of functions that they can approximate, the are often viewd as "black boxes." In other words, the get the job done, but it is extremely difficult to understand how the task was accomplished. The extraction of any kind of useful knowledge from the representation of the network is a hard problem.

In many applications, it is desirable to define a set of rules associated with the problem and solution pairs. Therefore, obtaining a neural network with an easily accessible set of rules based on its internal representation – weights and hidden unit activation patterns – is desired. Many approaches have been developed for extracting rules from a trained neural network. Decompostional algorithms extract rules from each unit in a network and combine them. Pedogogical algorithms generate samples from a trained network and induce rules from the samples. [8]
Towell and Shavlik describe two methods for extracting rules from a network. [7] The subset method described by Fu [4], searches for subsets of connections to a unit whose summed weight exceeds the bias of the unit. The M of N method deals with clusters of weights in equivalence classes. This algorithm addresses some of the problems inherent in the subset method. This method searches for rules of the form: If (M of N are true) then ___ . [7] towell These methods were all introduced by 1992. In 1994, Craven and Shavlik introduced a rule extraction

method based on sampling and queries. [2] Here, rule extraction itself is viewed as a learning task. In 1997, Setiono described a new method that required a trained neural network that has achieved a satisfactory performance level be pruned, thus reducing the network's complexity. [6] The RX-algorithm for extracting rules is presented. In 2000, Fu developed the DOMRUL system composed of a neural network and a rule extraction engine [3]. The activation function of the neural network is based on the Certainty Factor model of the MYCIN expert system. Also in 2000, Tsukimoto introduced an algorithm for rule extraction relying on the idea that units in the network are approximated by boolean functions. The polynomial algorithm then minimizes Euclidean distance. [8]

The hypothesis for this study is that a neural network using competitive distribution of error will yield a weight matrix that more clearly understandable by inspection than that of the corresponding weight matrix generated by the standard EBP network. The understandability is defined by the ability to extract rules from the more recognizable representation of the network. The hope is the this method will incorporate several of the ideas used in the previous methods. If weights in the trained network are close to zero, then these connections can be pruned. This is similar to [6]. If the network representation is such that hidden unit activation patterns are easily seen, then boolean valued rules can be generated by hand. In this case, the investigator can generate M of N or similar type rules as those described in [7].

2 Methods

The primary area of machine learning being evaluated is neural network learning using backpropagation.

2.1 Implementation

The program used to implement both the standard error backpropagation (EBP) neural network, and the neural network using the new competitive distribution of error (CDE) algorithm was done in C. A portion of the code is included in the appendix. The implementation follows the standard EBP algorithm with some added features to store the data generated by the execution of the program. The complete code is too lengthy to include, but is available from the author by email.

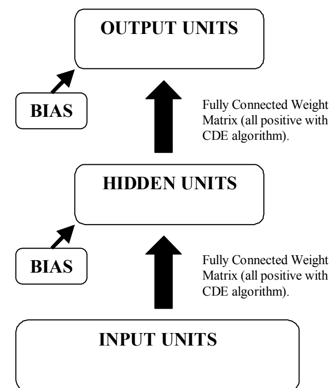

Figure 1: Basic Network Architecture.

Both algorithms operate with a sigmoid activation function with a slope that increases with time. This ensures that clear winners emerge in competition and that activations on fully on or off. Clearly this in an advantage when considering rule extraction, since units can be considered boolean valued and

rules in a boolean rule.

The CDE algorithm uses a feedforward competitive activation dynamic similar to the method used in Cho [1]. The network is required to have all positive weights as well as a bias to both the hidden and output units which may take a negative value. If during learning, a non-bias weight becomes negative, it is automatically set to some very small postive value. Initially during the feedforward process, activation is propagated as in standard EBP, here using a standard sigmoid activation function with assigned slope. However, a competition follows this intial propagation of forward activity so that winner nodes emerge.

The competition mechanism is taken from Cho [1]. First, a competition occurs at the hidden layer, where a node's activation at time t+1 is not only dependent on its net input, but also on its own activation according to equations (1a) and (1b), where f is some activation function (in this case sigmoid with steep slope). An addition parameter can also be added as an exponent to the activation to increase the competition.

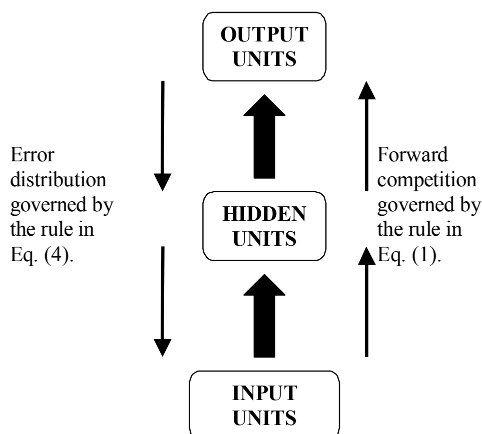

Figure 2: Schema of forward and backward competitions in the CDE algorithm.

(1a) $$in_{ji} = \frac{a_j w_{ji} a_i}{\sum_{k \in ResponseLayer} a_k w_{ki}}$$

(1b) $$a_i(t+1) = f(\sum in_{ji})$$

In this implementation, the competition occurs for a fixed number of time step iterations regardless of whether or not a "fixed state" was reached. Subsequently the competition occurs at the hidden layer in the same fashion to yield a winner node.

The backward competition is based on a competitive distribution of the error signals received by the hidden layer. First the delta values are computed and the hidden to output layer weights are adjusted using the standard EBP algorithm, where a_j is the activation of the jth output unit.

(2) $$\delta_j = (target - a_j)(a_j)(1 - a_j)$$

At time step zero, the delta values for the hidden units are also calculated using standard EBP.

(3) $$\delta_j = (\sum_{k \in O} \delta_k w_{kj})(a_j)(1 - a_j)$$

At this stage, the delta value for the set of hidden units can be represented in a diagram like Figure 3.

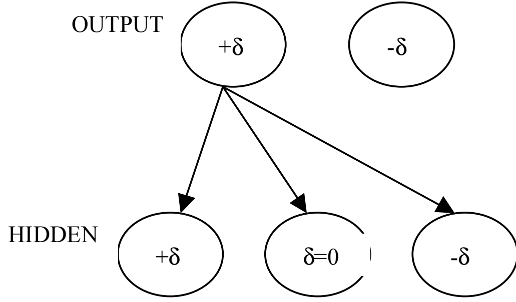

Figure 3: Summary of delta values.

At this point, it is desirable to have output units with positive delta values have more influence on those hidden units with positive delta values and less influence on the hidden units with negative delta values. Negative output delta values are considered analogously. The distribution of the output delta values is defined by (3) in standard EBP. Now, we introduce a competitve mechanism to distribute the output delta values to the "more deserving" hidden units. Below δ_i is the error signal at the ith output unit, Δ_j is the previous iteration sum of error signal at hidden unit j, and s_i is the sign of δ_i. An additional factor can also be added to increase or decrease competition.

(4a) $$back_{jk} = \frac{\exp(s_i \Delta_j) w_{ij}}{\sum_k \exp(s_i \Delta_k) w_{ik}} \delta_i$$

(4b) $$\delta_j = \sum_{k \in O} back_{jk}$$

This competition continues for several iterations and is built in as a user specifed parameter in the program. After the competition, all weights in the network are changed as in standard EBP.

(5) $$\Delta w_{ji} = \eta \delta_j a_i$$

2.2 Data

The data for this experiment was constructed by hand, based on what baseball pitch "is" (according to me, not Roger Clemens) most appropriate given the count. The data is oversimplified, with only three possible pitches and the number of outs not taken into consideration. The input is a binary sequence of 7 input units, representing 0 balls through 3 balls and 0 strikes through 2 strikes. For any given pattern, the pitch count is represented so that the number of balls and strikes receive a one in the corresponding input units and zeros for all other input units. The data was designed so that a set of rules was known a priori. The complete data can be found in the appendix. The rules that can be generated by the data can be expressed as follows.

Table 1: Rules in the Data

ANTECEDENTS	CONSEQUENCE
0 Balls, 0 Strikes	Fastball
# Balls < # Strikes	Curveball
# Strikes < # Balls	Fastball
# Balls = # Strikes (non-zero)	Offspeed

3 Results

The program was executed, generating trained neural networks using both standard EBP and the CDE algorithm.

3.1 Performance

Both networks were able to classify the twelve training examples correctly. They were evaluated by two independent measurements. The first is an error

metric, which measures the root mean square error between the actual output values generated by the network and the target correct values (RMSE). The second metric is a correctness measure; simply the percentage of output units that are correctly on or off (PCT).

All parameters which are non-essential to the particular algorithm used were kept constant so that a valid comparison between the two methods a could be made. These include the number of hidden units, the initial activation function parameters, and the learning rate. Two simulations are compared below and complete logs of the two simulations can be found in the appendix.

Table 2: Performance Comparison

	EBP	CDE
Epochs Trained	55	70
RMSE	0.0283	0.0243
PCT	100	100

Not only did both algorithms classify the examples correctly, they are remarkably similar in running time and classification accuracy.

3.2 Weights and Rules

Recall the hypothesis of this study: A neural network using competitive distribution of error will yield a weight matrix that more clearly understandable by inspection than that of the corresponding weight matrix generated by the standard EBP network. The understandability is defined by the ability to extract rules from the more recognizable representation of the network.

Since the data used in this experiment was designed to represent rules that were known ahead of time, it is easy to see if the results of the simulation conform to the rules is some easily recognized fashion. Let us consider each method in turn. The weight matrices for both trained networks can be found in the appendix.

3.2.1 Standard EBP

The standard EBP network used weights and biases that can each take on positive or negative values. This in itself complicates the ability determine which weights contribute to the activation pattern of the hidden units. There is no obvious pattern of weights in the matrix. For example, none of the weights are close to zero; many of the weights are of a similar magnitude; there does not appear to be any sequence increasing or decreasing weights across input units that might conform to the predetermined rules.

In addition, the hidden unit activation pattern is also distributed. Therefore, all hidden units then contribute in some fashion to the correctly generated ouput. Often there is no hidden unit or set of hidden units that can identified with a particular input sequence pattern to correspond to a known rule. Overall, the neural network representation is difficult to understand for this simple example, despite the fact that it learns the correct classifications.

3.2.2 CDE

The neural network employing the competitive distribution of error algorithm used all positive weights. In this fashion, any weight other than the bias contributes positively (or zero) to the net input of hidden units. The competitive distribution of error more clearly separates weight changes applied

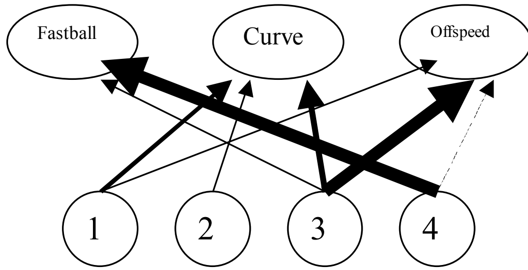

Figure 4: Hidden units labeled 1-4 with weights to output units. The weights are all positive and thus their magnitudes are represented by the relative size of the arrow.

to the network, and thus allows rules to be seen more easily.

The weight matrix for the CDE algorithm contains many zeros. These connections between units with a zero weight can subsequently be ignored in all analysis. In addition, it is the case in several instances that for a given input unit, all connections except one have a zero valued weight. Thus, this is clear antecedent to a rule: If input unit A, then hidden unit B.

Specifically, we can easily see a rule stating: If (3 Balls OR 0 Strikes) Then Hidden Unit 5 is On. If (Hidden Unit 5) Then Fastball. This rule exactly corresponds to the data. The network has learned to relate a particular hidden unit with any case where there are 3 Balls or 0 Strikes. Other similar rules can also be generated to match the data.

4 Conclusion

The CDE algorithm was able to learn the simple baseball pitch problem. The weight matrix generated was far more understandble to the human eye, with no computation needed. Rules could be generated by inspection that coorespond to the known rules which were used to generate the data. The method proved to be analogous to pruning, as many weights were zero-valued and could be ignored in analyis. The competitve mechanism also allowed boolean functions to be used to generate the rules, in other words hidden units were clearly on in each case.

While this method is relatively rudimentary, there is promise that it will generalize to other problems, most notably the XOR and parity problems, as well as other pattern classification and real-valued function approximation problems.

References

[1] Cho, Sungzoon. (1992) *Learning Competition and Cooperation*. University of Maryland Technical Report Series, CS-TR-2887.

[2] Craven, M.W. and Jude Shavlik. (1994) *Using sampling and queries to extract rules from trained neural networks*. Proceedings of the 11th International Conference on Machine Learning.

[3] Fu, Li Min. (2000) *The Application of Certainty Factors to Neural Computing for Rule Discovery*. IEEE Transactionson Neural Networks. vol. 11, no. 3, 647-657.

[4] Fu, L. (1991) *Rule Learning by Searching on Adapted Nets*. Proc. of 9th National Conference on Artificial Intelligence, 590-595.

[5] Setiono, Rudy. (1997) *A penalty-function approach for pruning*

feedforward neural networks. Neural Computation. vol. 9, no. 1, 185-204.

[6] Setiono, Rudy. (1997) *Extracting rules from neural networks by pruning and hidden unit splitting*. Neural Computation. vol. 9, no. 1, 205-225.

[7] Towell, Geoffrey G. and Jude W. Shavlik. (1993) *Extracting Refined rules from Knowledge-based neural networks*. Machine Learning. vol. 13, 71-101.

[8] Tsukimoto, Hiroshi. (2000) *Extracting Rules from trained neural networks.* IEEE Transactions on Neural Networks. vol. 11, no. 2, 377-389.

APPENDIX A: Code Fragments

1) This function propagates activity forward in the network. Competitive activation is used if the flag COMP_ACT is true.

```
/*----------------------------------------------------*/
private void
forwardPropagate( boolean prnt ) {

        int i, j, k, numEqual;
        float compSum;

   /* ------ zero hidden layer inputs ------ */
   zeroMemory( hiddenInput,  sizeof(hiddenInput[0]) * numHiddenUnits );

/* Print pattern */
if ((epochsTrained % PRINT_TEST_FREQUENCY == 0) && prnt){
        fprintf(actFile, "\nPattern: ");
        for (j = 0; j < numInputUnits; j++)
                fprintf (actFile, "%3.1f ", iValue[j]);
}

   /* ------ input to hidden hidden layer ------ */

        for ( i=0; i<numHiddenUnits; i++ ) {
                if (USE_HBIAS)
                        hiddenInput[i] = hBiasWt[i];
                else
                        hiddenInput[i] = 0.0;

                for ( j=0; j<numInputUnits; j++ )
                        hiddenInput[i] += hiddenIweight[i][j] * iValue[j];
        }

        for (i = 0; i < numHiddenUnits; i++)
                hiddenValue[i] = hiddenTransferFn(hiddenInput[i]);

        if ((epochsTrained % PRINT_TEST_FREQUENCY == 0) && prnt){
                fprintf(actFile, "\n\nHidden Input and Activation within forward propagate, Epoch %3d,
time  0.\n", epochsTrained);
                for (j = 0; j < numHiddenUnits; j++)
                        fprintf (actFile, "%5.3f  ", hiddenInput[j]);
                fprintf(actFile, "\n");
                for(j = 0; j < numHiddenUnits; j++)
                        fprintf (actFile, "%5.3f  ", hiddenValue[j]);
                fprintf (actFile, "\n");
        }

        if ( COMP_ACT && (COMP_ACT_HID_ITER > 0) ) {

                for (k = 0; k < 10*COMP_ACT_HID_ITER; k++){
                        zeroMemory( hiddenInput,  sizeof(hiddenInput[0]) * numHiddenUnits );
                        for (i = 0; i < numInputUnits; i++){
```

```
					compSum = 0.0;
					for ( j = 0; j < numHiddenUnits; j++){
						if(USE_HBIAS) compSum += hBiasWt[j] *
(pow(hiddenValue[j], P_ACT) + Q_ACT);
						compSum += hiddenIweight[j][i] * (pow(hiddenValue[j],
P_ACT) + Q_ACT);
					}
					for ( j = 0; j < numHiddenUnits; j++){
						if(USE_HBIAS)
							hiddenInput[j] += hBiasWt[j] * (pow(hiddenValue[j],
P_ACT) + Q_ACT) / compSum;
						hiddenInput[j] += hiddenIweight[j][i] * iValue[i] *
(pow(hiddenValue[j], P_ACT) + Q_ACT) / compSum;
					}
				}

				numEqual = 0;
				for ( j = 0; j < numHiddenUnits; j++ ){
					tempH[j] = hiddenValue[j];
					hiddenValue[j] += 0.1 * (hiddenTransferFn(hiddenInput[j]) - BETA *
hiddenValue[j]);
					if ( (abs(tempH[j] - hiddenValue[j])) < MIN_ACT_DIFF)
						numEqual++;
				}

				if (numEqual == numHiddenUnits)
					break;

			}

			if ((epochsTrained % PRINT_TEST_FREQUENCY == 0) && prnt){
				fprintf(actFile, "\n\nHidden Input and Activation within forward propagate,
Epoch %3d, time %2d.\n", epochsTrained, k);
			for (j = 0; j < numHiddenUnits; j++)
				fprintf (actFile, "%5.3f  ", hiddenInput[j]);
			fprintf(actFile, "\n");
			for(j = 0; j < numHiddenUnits; j++)
				fprintf (actFile, "%5.3f  ", hiddenValue[j]);
			fprintf (actFile, "\n");
			}

		}

   /* ------ propagate to output layer ------ */

		for ( i=0; i<numOutputUnits; i++ ) {
			if (USE_OBIAS)
				oInput[i] = oBiasWt[i];
			else
				oInput[i] = 0.0;
			for ( j=0; j<numHiddenUnits; j++ )
	oInput[i] += oHiddenWeight[i][j] * hiddenValue[j];
		}
		for ( i=0; i<numOutputUnits; i++ )
			oValue[i] = oTransferFn( oInput[i] );
```

```
	if ((epochsTrained % PRINT_TEST_FREQUENCY == 0) && prnt){
		fprintf(actFile, "\n\nOutput Input and Activation within forward propagate, Epoch %3d, 
time  0.\n", epochsTrained);
		for (j = 0; j < numOutputUnits; j++)
			fprintf (actFile, "%5.3f  ",oInput[j]);
		fprintf(actFile, "\n");
		for(j = 0; j < numOutputUnits; j++)
			fprintf (actFile, "%5.3f  ", oValue[j]);
		fprintf (actFile, "\n");
	}

	if ( COMP_ACT && (COMP_ACT_OUT_ITER > 0) ) {

		for (k = 0; k < 10*COMP_ACT_OUT_ITER; k++){
			zeroMemory( oInput,  sizeof(oInput[0]) * numOutputUnits );
			for (i = 0; i < numHiddenUnits; i++){
				compSum = 0.0;
				for ( j = 0; j < numOutputUnits; j++){
					if(USE_OBIAS) compSum += oBiasWt[j] * (pow(oValue[j], 
P_ACT) + Q_ACT);
					compSum += oHiddenWeight[j][i] * (pow(oValue[j], P_ACT) 
+ Q_ACT);
				}

				for ( j = 0; j < numOutputUnits; j++){
					if(USE_OBIAS)
						oInput[j] += oBiasWt[j] * (pow(oValue[j], P_ACT) + 
Q_ACT) / compSum;
					oInput[j] += oHiddenWeight[j][i] * hiddenValue[i] * 
(pow(oValue[j], P_ACT) + Q_ACT) / compSum;
				}
			}

			numEqual = 0;
			for ( j = 0; j < numOutputUnits; j++ ){
				tempO[j] = oValue[j];
				oValue[j] += 0.1 * (oTransferFn(oInput[j]) - BETA * oValue[j]);
				if ( (abs(tempO[j] - oValue[j])) < MIN_ACT_DIFF )
					numEqual++;
			}

			if (numEqual == numOutputUnits)
				break;

		}

		if ((epochsTrained % PRINT_TEST_FREQUENCY == 0) && prnt){
			fprintf(actFile, "\n\nOutput Input and Activation within forward propagate, 
Epoch %3d, time  %2d.\n", epochsTrained, k);
		for (j = 0; j < numOutputUnits; j++)
			fprintf (actFile, "%5.3f  ",oInput[j]);
		fprintf(actFile, "\n");
		for(j = 0; j < numOutputUnits; j++)
			fprintf (actFile, "%5.3f  ", oValue[j]);
		fprintf (actFile, "\n");
		}
```

```
        }
}
```

2) This function accumlates the error signals or delta values used in the backpropagation of errors. The CDE algorithm is used if the COMP_ERR and SOFT_MAX flags are true.

```
/*---------------------------------------------------*/
private void
accumulateDeltaWeights( int row ) {

   float   adjError, compSum;
   int     i, j, k;

   /* ------ accumulate hidden->output delta weights ------ */
   zeroMemory( hiddenError,  sizeof(hiddenError[0])  * numHiddenUnits );
   zeroMemory( outputError,  sizeof(outputError[0])  * numOutputUnits );

        /* Calculate delta_O_i's = (c_i - a_i)*f'(a_i) */
        for ( i=0; i<numOutputUnits; i++ ) {
                outputError[i] = (category[row][i]-oValue[i])*( oTransferFnPrime(oValue[i]) + LEAK );
        }

        for (i = 0; i < numOutputUnits; i++){
                if (RPROP)
                        adjError = outputError[i];
                else
                        adjError    = learningRate * outputError[i];
                for ( j=0; j<numHiddenUnits; j++ ) {
                        oHiddenDeltaWeight[i][j] += adjError    * hiddenValue[j];
                        hiddenError[j]          += outputError[i] * oHiddenWeight[i][j];
                }

                if (USE_OBIAS)
                        oBiasDeltaWt[i] += adjError;

        }

        for (  j = 0; j < numHiddenUnits; j++){
                /* Compute delta_H_j's = (SUM(delta_O_i's*w_ij))*f'(a_j) */
                hiddenError[j] *= ( hiddenTransferFnPrime( hiddenValue[j] ) + LEAK );
        }

        if (COMP_ERROR && SOFT_MAX) {

if (epochsTrained % PRINT_TEST_FREQUENCY == 0){
        /* Print pattern */
        fprintf(errFile, "\n\nPattern:  ");
        for (j = 0; j < numInputUnits; j++)
                fprintf (errFile, "%3.1f ", patterns[row][j]);
        /* Print Category */
        fprintf(errFile, "\nCategory: ");
        for (j = 0; j < numOutputUnits; j++)
                fprintf(errFile, "%3.1f ", category[row][j]);
```

```
}

                if (epochsTrained % PRINT_TEST_FREQUENCY == 0){
                        fprintf( errFile, "\n\nHidden Error Value, Epoch %3d, time 0.\n",
epochsTrained);
                        for (j = 0; j < numHiddenUnits; j++)
                                fprintf (errFile, "%7.3f ", hiddenError[j]);
                }

                for (k = 0; k < COMP_ERR_ITER; k++){

                        for ( j = 0; j < numHiddenUnits; j++ )
                                hiddenErrorSaved[j] = hiddenError[j];

                        zeroMemory( hiddenError, sizeof(hiddenError[0]) * numHiddenUnits );

                        for (i = 0; i < numOutputUnits; i++){
                                compSum = 0.0;
                                for ( j = 0; j < numHiddenUnits; j++)
                                        compSum += exp(P_ERR * sign(outputError[i]) *
hiddenErrorSaved[j]) * oHiddenWeight[i][j];
                                for ( j = 0; j < numHiddenUnits; j++)
                                        hiddenError[j] += (exp(P_ERR * sign(outputError[i]) *
hiddenErrorSaved[j]) * oHiddenWeight[i][j] * outputError[i]) / compSum;
                        }

                }

                if (epochsTrained % PRINT_TEST_FREQUENCY == 0){
                        fprintf( errFile, "\n\nHidden Error Value, Epoch %3d, time %3d.\n",
epochsTrained, k);
                        for (j = 0; j < numHiddenUnits; j++)
                                fprintf (errFile, "%7.3f ", hiddenError[j]);
                }

        } // end if COMP_ERROR


  /* ------ accumulate hidden input->hidden delta weights ------ */
  for ( i=0; i<numHiddenUnits; i++ ) {
    if (RPROP)
                        adjError = hiddenError[i];
                else
                        adjError = learningRate * hiddenError[i];

    for ( j=0; j<numInputUnits; j++ )
                        hiddenIdeltaWeight[i][j] += adjError * iValue[j];

                if (USE_HBIAS)
                        hBiasDeltaWt[i] += adjError;

  }

}
```

APPENDIX B: Data

0B	1B	2B	3B	0S	1S	2S	Fastball	Curve	Offspeed
		X		X			X		
X				X			X		
	X			X			X		
			X			X	X		
		X			X				X
X					X			X	
	X					X		X	
	X				X				X
			X	X			X		
			X		X			X	
		X				X	X		
X						X	X		

APPENDIX C: Simulation Results

Standard EBP:

```
** LOG FILE **

11:49WedAug29
Input File: baseball.txt
Run ID #: 45

Inputs: 7   Outputs: 3   Hidden Units: 5

Using initial weights from file net0.0041_11:49WedAug29.

Using logistic activation function with slope  1.0 and
shift 0.50.
Activation slope increases for the first 100 epochs by
0.09.

Using bias to hidden layer.

Using bias to output layer.

Learning rate:  0.200

Using incremental learning mode.

    0 epochs:  rmse = 0.5467  pct =   0.00
   10 epochs:  rmse = 0.4222  pct =   0.00
   20 epochs:  rmse = 0.3065  pct =  16.67
   30 epochs:  rmse = 0.2274  pct =  58.33
   40 epochs:  rmse = 0.1581  pct =  86.11
   50 epochs:  rmse = 0.0446  pct =  91.67
   55 epochs:  rmse = 0.0283  pct = 100.00

Network saved to net.0045_12:46WedAug29

Finished simulation at 12:46WedAug29.
```

```
***** Weights after 55 training epochs *****

-----    FROM INPUT TO HIDDEN UNITS    -----
 0.54  0.30  0.19 -0.10  0.36  0.36  0.09 -0.08
-0.06  0.86  0.40 -0.15 -0.65 -0.65  0.13  0.68
 0.30 -0.65  0.23  1.06  0.44  0.29  0.52 -0.42
-0.82  0.14  0.24  0.22  0.28  0.04  0.17  0.44
 0.01 -0.24 -0.42  0.60  1.16  1.19  0.35 -0.56

-----    FROM HIDDEN TO OUTPUT UNITS   -----
 0.09  0.19 -1.27  0.10  0.10  1.40
 0.11  0.09  1.22 -0.83  0.38 -0.88
 0.07 -0.24  0.06  1.06  0.34 -1.03

*** Hidden Activation of network net.0045_12:46WedAug29.
***


*** Using input file baseball.txt. ***

*** Time Stamp: 14:32WedSep05. ***


Using competitive activation dynamics (Cho),
with 20 iters at HL and 30 iters at OL,
and using P = 1.50, Q = 0.00, and act-decay = 1.00.

Using logistic activation function with slope  1.0 and
shift 0.50.
Activation slope increases for the first 100 epochs by
0.09.

Using competitive errors, 20 iterations, P = 1.50 with SOFT
MAX.

Using bias to hidden layer.

Using bias to output layer.


PATTERN  1: 0.0 0.0 1.0 0.0 1.0 0.0 0.0
HIDDEN ACT: 0.398 0.333 0.553 0.398 0.694

PATTERN  2: 1.0 0.0 0.0 0.0 1.0 0.0 0.0
HIDDEN ACT: 0.574 0.530 0.312 0.415 0.554

PATTERN  3: 0.0 1.0 0.0 0.0 1.0 0.0 0.0
```

```
HIDDEN ACT: 0.539 0.397 0.545 0.467 0.429

PATTERN  4: 0.0 0.0 0.0 1.0 0.0 0.0 1.0
HIDDEN ACT: 0.413 0.293 0.517 0.338 0.792

PATTERN  5: 0.0 0.0 1.0 0.0 0.0 0.0 1.0
HIDDEN ACT: 0.384 0.302 0.693 0.335 0.663

PATTERN  6: 1.0 0.0 0.0 0.0 0.0 1.0 0.0
HIDDEN ACT: 0.444 0.713 0.374 0.432 0.403

PATTERN  7: 0.0 1.0 0.0 0.0 0.0 0.0 1.0
HIDDEN ACT: 0.395 0.781 0.359 0.528 0.294

PATTERN  8: 0.0 1.0 0.0 0.0 0.0 1.0 0.0
HIDDEN ACT: 0.434 0.530 0.598 0.468 0.349

PATTERN  9: 0.0 0.0 0.0 1.0 1.0 0.0 0.0
HIDDEN ACT: 0.431 0.320 0.431 0.398 0.773

PATTERN 10: 0.0 0.0 1.0 0.0 0.0 1.0 0.0
HIDDEN ACT: 0.380 0.381 0.695 0.416 0.497

PATTERN 11: 0.0 0.0 0.0 1.0 0.0 1.0 0.0
HIDDEN ACT: 0.417 0.353 0.532 0.424 0.651

PATTERN 12: 1.0 0.0 0.0 0.0 0.0 0.0 1.0
HIDDEN ACT: 0.395 0.841 0.305 0.466 0.315
```

CDE Algorithm:

```
** LOG FILE **

11:49WedAug29
Input File: baseball.txt
Run ID #: 44

Inputs: 7   Outputs: 3   Hidden Units: 5

Using initial weights from file net0.0041_11:49WedAug29.

Using competitive activation dynamics (Cho),
with 20 iters at HL and 30 iters at OL,
and using P = 1.50, Q = 0.00, and act-decay = 1.00.

Using logistic activation function with slope  1.0 and
shift 0.50.
Activation slope increases for the first 100 epochs by
0.09.

Using competitive errors, 20 iterations, P = 1.50 with SOFT
MAX.

Using bias to hidden layer.

Using bias to output layer.

Learning rate:  0.200

Using incremental learning mode.

    0 epochs:  rmse = 0.5240  pct =   0.00
   10 epochs:  rmse = 0.5229  pct =   5.56
   20 epochs:  rmse = 0.5149  pct =   8.33
   30 epochs:  rmse = 0.4900  pct =   8.33
   40 epochs:  rmse = 0.4348  pct =  19.44
   50 epochs:  rmse = 0.3892  pct =  69.44
   60 epochs:  rmse = 0.4548  pct =  77.78
   70 epochs:  rmse = 0.0256  pct = 100.00
   70 epochs:  rmse = 0.0243  pct = 100.00

Network saved to net.0044_12:36WedAug29

Finished simulation at 12:39WedAug29.
```

```
***** Weights after 70 training epochs *****

-----    FROM INPUT TO HIDDEN UNITS    -----
-4.35  0.12  0.11  0.01  0.00  0.00  0.06  0.43
-3.61  0.00  0.06  0.00  0.00  0.00  0.01  0.06
-4.33  0.00  0.10  0.05  0.00  0.00  0.12  0.40
-4.86  0.34  0.08  0.00  0.00  0.00  0.00  0.43
-4.41  0.00  0.00  0.20  0.44  1.31  0.09  0.00

-----    FROM HIDDEN TO OUTPUT UNITS   -----
-3.69  0.00  0.00  0.02  0.00  0.24
-9.18  0.09  0.02  0.15  0.00  0.00
-8.33  0.03  0.00  0.22  0.00  0.01
```