Laboratory Assignment #3: Back Propagation Learning

Bryan Guner, James Martinez, and Olivia Shanley
Anthony Deese, Ph. D.
Artificial Neural Networks - ELC 470-01
The College of New Jersey
Department of Electrical and Computer Engineering (ECE)
03/22/2017

Intro and Procedure:

For this experiment our team was tasked with adapting a Matlab sample code to create an artificial neural network that emulates the XOR Boolean function. This task could not be achieved with the single-layer Perceptron because the XOR function violates linear separability. The resulting network received two inputs, however, it employed x1 and the absolute difference between x1 and x2, instead of x1 and x2. The "new" network was designed with three neurons in the hidden layer, as opposed to the two used in the given example. The output layer remained exactly the same as in the example. The goal was for the ANN to generate two outputs, XOR and its inverse !XOR. The neuron bias/threshold values provided in the lab were not to be changed, however, we decided to rearrange the synapse connections between the other neurons. All neurons remained connected to the network in some manner.

In order to achieve said ANN, one must first Initialize the synapse weights between the neurons and the bias weights. Further we must define an update coefficient and create a loop to perform iterations of network training. At this point we conduct forward propagation which consists of generating a random training sample, initializing the output of neurons 1 & 2, cycling through non-input neurons,initializing the weighted sum of neurons to the threshold value (in this case 1) as input to the activation function, cycling through adjacent neurons, applying the activation function and rounding the outputs to 1 or 0 depending on proximity. Then we perform back

propagation on the output layer which consists of calculating the partial derivatives, recording the old weights, updating the synapse weights and defining the error. This back propagation is then repeated on the hidden layer.

Matlab Code:

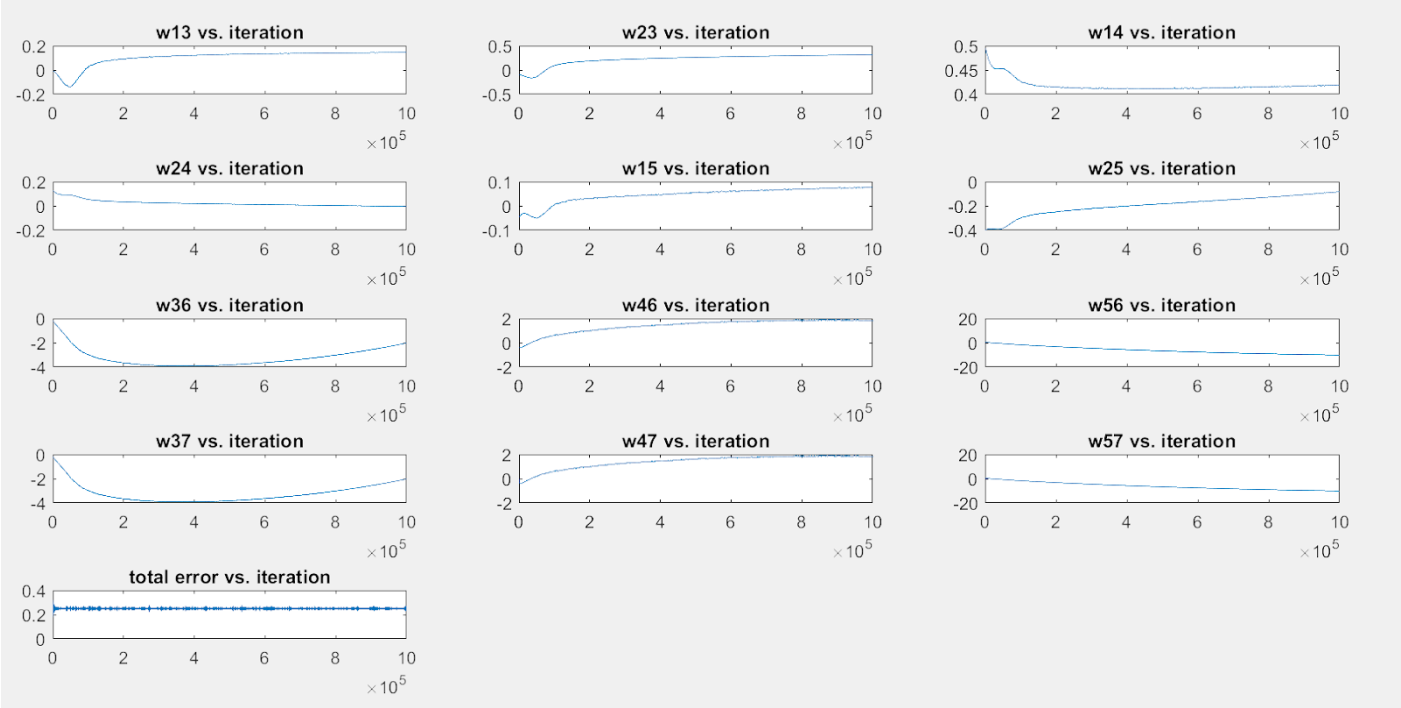
```
NOTE: You may need to run the learning algorithm several times in order
% to achieve convergence.  The learning algorithm is very sensitive to
% the randomized initial conditions.

% ACTION: Defining training data for XOR gate.
% Column #1 is x1.
% Column #2 is abs(x1-x2).
% Column #3 is XOR output.
% Column #4 is not XOR (XOR!) output.
matrixTraining = [0 0 0 1; 0 1 1 0; 1 1 1 0; 1 0 0 1];

% ACTION: Initialize synapse weights between neurons.
matrixW = [0 0 0 0 0 0 0; 0 0 0 0 0 0 0; (rand()-0.5) (rand()-0.5) 0 0 0 0 0; (rand()-0.5) (rand()-0.5) 0 0 0 0 0; (rand()-0.5)
(rand()-0.5) 0 0 0 0 0; 0 0 (rand()-0.5) (rand()-0.5) (rand()-0.5) 0 0; 0 0 (rand()-0.5) (rand()-0.5) (rand()-0.5) 0 0]; % input
weights only
% this is a sample solution... matrixW = [0 0 0 0 0 0 0; 0 0 0 0 0 0 0; -4.81305 4.537743 0 0 0 0 0; -3.52158 3.452617 0 0 0 0 0; 0 0
3.698052 -3.38708 0 0; 0 0 -3.72364 3.422721 0 0]; % input weights only

% ACTION: Initialize bias weights.
arrayW0 = [0 0 (rand()-0.5) (rand()-0.5) (rand()-0.5) (rand()-0.5) (rand()-0.5)];
% this is a sample solution... arrayW0 = [0 0 -3.08689 1.412882 1.463913 -1.47856 ];

% ACTION: Initializ array to contain output of individual neurons.
yk = [-99 -99 -99 -99 -99 -99 -99];

% ACTION: Extract training data to arrays from matrixTraining.
x1 = matrixTraining(:,1); % x1
xdiff = matrixTraining(:,2); % x1-x2
y6ID = matrixTraining(:,3); % XOR output
y7ID = matrixTraining(:,4); % XOR! output

% ACTION: Define update coefficient.
alpha = 0.005;

% ACTION: Create loop to perform iterations.
for kIteration = 1:1000000

   % FORWARD PROPAGATION ----------------------------------------------

   % ACTION: Randomize training sample (1-4) to be used.
   kTraining = floor(4*rand())+1;

   % ACTION: Initialize output of neurons 1 and 2 as defined by training
   % data sample associated with kTraining.
   yk(1) = x1(kTraining);
   yk(2) = xdiff(kTraining);

   % ACTION: Cycle through all non-input neurons.
   for k01 = 3:7

      % ACTION: Initialize weighted sum to bias/threshold value.
```

```
    xk = arrayW0(k01)*1;

    % ACTION: Cycle through all potential adjacent neurons.
    for k02 = 1:7

        % ACTION: Created weighted sum as input to activation function
        % for neuron k01 by adding new elements in loop.
        xk = xk + yk(k02)*matrixW(k01,k02);

    end

    % ACTION: Apply activation function.
    activation_output = 1/(1+exp(-xk));

    % ACTION: Assume that outputs "close" to 1 or 0, do take on that
    % value.
    if(k01 == 6 || k01 == 7)
        if(activation_output >= 0.75)
            activation_output = 1;
        elseif(activation_output < 0.25)
            activation_output = 0;
        end
    end

    % ACTION: Save activation_output to yk as appropriate.
    yk(k01) = activation_output;
end

% ACTION: Calculate the output errors at neurons 6 and 7.
e6 = 0.5*(y6ID(kTraining) - yk(6))^2;
e7 = 0.5*(y7ID(kTraining) - yk(7))^2;

% ACTION: Record total error.
eTotal(kIteration) = e6 + e7;

% APPLY BACK PROPAGATION TO OUTPUT LAYER ------------------------------

% ACTION: Calculate partial derivatives.
dError_dw36 = (-1)*(y6ID(kTraining)-yk(6))*yk(5)*(1-yk(6))*yk(3);
dError_dw46 = (-1)*(y6ID(kTraining)-yk(6))*yk(6)*(1-yk(6))*yk(4);
dError_dw56 = (-1)*(y6ID(kTraining)-yk(6))*yk(6)*(1-yk(6))*yk(5);
dError_dw37 = (-1)*(y7ID(kTraining)-yk(7))*yk(7)*(1-yk(7))*yk(3);
dError_dw47 = (-1)*(y7ID(kTraining)-yk(7))*yk(7)*(1-yk(7))*yk(4);
dError_dw57 = (-1)*(y7ID(kTraining)-yk(7))*yk(7)*(1-yk(7))*yk(5);

% ...don't forget biases.
dError_dw06 = (-1)*(y6ID(kTraining)-yk(6))*yk(6)*(1-yk(6))*1;
dError_dw07 = (-1)*(y7ID(kTraining)-yk(7))*yk(7)*(1-yk(7))*1;

% ACTION: Record old weights.
record_w36(kIteration) = matrixW(6,3);
record_w46(kIteration) = matrixW(6,4);
record_w56(kIteration) = matrixW(6,5);
record_w37(kIteration) = matrixW(7,3);
```

```
    record_w47(kIteration) = matrixW(7,4);
    record_w57(kIteration) = matrixW(7,5);
    record_w06(kIteration) = arrayW0(6);
    record_w07(kIteration) = arrayW0(7);

    % ACTION: Update weights.

    matrixW(6,3) = matrixW(6,3) - alpha*dError_dw36;
    matrixW(6,4) = matrixW(6,4) - alpha*dError_dw46;
    matrixW(6,5) = matrixW(6,5) - alpha*dError_dw56;
    matrixW(7,3) = matrixW(6,3) - alpha*dError_dw37;
    matrixW(7,4) = matrixW(6,4) - alpha*dError_dw47;
    matrixW(7,5) = matrixW(6,5) - alpha*dError_dw57;

    % ... don't forget biases.
    arrayW0(6) = arrayW0(6) - alpha*dError_dw06;
    arrayW0(7) = arrayW0(7) - alpha*dError_dw07;

    % ACTION: Define epsilon values.
    epsilon6 = (-1)*(y6ID(kTraining)-yk(6))*yk(6)*(1-yk(6));
    epsilon7 = (-1)*(y7ID(kTraining)-yk(7))*yk(7)*(1-yk(7));
    % APPLY BACK PROPAGATION TO HIDDEN LAYER ------------------------------
       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%55
    % ACTION: Calculate partial derivatives.

    dError_dw13 = (epsilon6*matrixW(6,3) + epsilon7*matrixW(7,3))*yk(3)*(1-yk(3))*yk(1);
    dError_dw23 = (epsilon6*matrixW(6,3) + epsilon7*matrixW(7,3))*yk(3)*(1-yk(3))*yk(2);
    dError_dw14 = (epsilon6*matrixW(6,4) + epsilon7*matrixW(7,4))*yk(4)*(1-yk(4))*yk(1);
    dError_dw24 = (epsilon6*matrixW(6,4) + epsilon7*matrixW(7,4))*yk(4)*(1-yk(4))*yk(2);
    dError_dw15 = (epsilon6*matrixW(6,5) + epsilon7*matrixW(7,5))*yk(5)*(1-yk(5))*yk(1);
    dError_dw25 = (epsilon6*matrixW(6,5) + epsilon7*matrixW(7,5))*yk(5)*(1-yk(5))*yk(2);
    % ...don't forget biases.
    dError_dw03 = (epsilon6*matrixW(6,3) + epsilon7*matrixW(7,3))*yk(3)*(1-yk(3))*1;
    dError_dw04 = (epsilon6*matrixW(6,4) + epsilon7*matrixW(7,4))*yk(4)*(1-yk(4))*1;
    dError_dw05 = (epsilon6*matrixW(6,5) + epsilon7*matrixW(7,5))*yk(5)*(1-yk(5))*1;

    % ACTION: Record old weights.
    record_w13(kIteration) = matrixW(3,1);
    record_w23(kIteration) = matrixW(3,2);
    record_w14(kIteration) = matrixW(4,1);
    record_w24(kIteration) = matrixW(4,2);
    record_w15(kIteration) = matrixW(5,1);
    record_w25(kIteration) = matrixW(5,2);
    record_w03(kIteration) = arrayW0(3);
    record_w04(kIteration) = arrayW0(4);
    record_w05(kIteration) = arrayW0(5);
    % ACTION: Update weights.

    matrixW(3,1) = matrixW(3,1) - alpha*dError_dw13;
    matrixW(3,2) = matrixW(3,2) - alpha*dError_dw23;
    matrixW(4,1) = matrixW(4,1) - alpha*dError_dw14;
    matrixW(4,2) = matrixW(4,2) - alpha*dError_dw24;
    matrixW(5,1) = matrixW(5,1) - alpha*dError_dw15;
```

```
    matrixW(5,2) = matrixW(5,2) - alpha*dError_dw25;
    % ... don't forget biases.
    arrayW0(3) = arrayW0(3) - alpha*dError_dw03;
    arrayW0(4) = arrayW0(4) - alpha*dError_dw04;
    arrayW0(5) = arrayW0(5) - alpha*dError_dw05;
end

%xTemp = linespace(-5,5);
subplot(5,3,1)
plot(record_w13)
title('w13 vs. iteration')
subplot(5,3,2)
plot(record_w23)
title('w23 vs. iteration')

subplot(5,3,3)
plot(record_w14)
title('w14 vs. iteration')
subplot(5,3,4)
plot(record_w24)
title('w24 vs. iteration')

subplot(5,3,5)
plot(record_w15)
title('w15 vs. iteration')
subplot(5,3,6)
plot(record_w25)
title('w25 vs. iteration')

subplot(5,3,7)
plot(record_w36)
title('w36 vs. iteration')
subplot(5,3,8)
plot(record_w46)
title('w46 vs. iteration')
subplot(5,3,9)
plot(record_w56)
title('w56 vs. iteration')

subplot(5,3,10)
plot(record_w37)
title('w37 vs. iteration')
subplot(5,3,11)
plot(record_w47)
title('w47 vs. iteration')
subplot(5,3,12)
plot(record_w57)
title('w57 vs. iteration')

subplot(5,3,13)
plot(eTotal)
title('total error vs. iteration')
```

Graphs



Graphs of Weights and Error vs Iteration

Conclusion:

Backpropagation works faster than earlier approaches to machine learning, making it possible to use neural nets to solve problems which had previously been impractical or even impossible (i.e. linearly inseparable) .At the core of the algorithm is the expression for the partial derivative of the cost function with respect to any weight or bias in the network. The expression illustrates how quickly the cost changes when we change the weights and biases.The backpropagation algorithm seeks the minimum of the error function in the weight space using a method called gradient descent. The combination of weights that minimizes the error function is considered to be a solution. Since this method calls for calculating the gradient of the error function at each iteration,the error function must be both continuous and differentiable. because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too, we have to use a kind of activation function other than the step function used in previous assignments. In our case we used the sigmoid function defined by the expression 1/(1+exp(-xk)).