Thomas Nabelek
September 22, 2016

ECE 7870
Project 1 – Backpropagation

1) Introduction

The backpropagation algorithm is a well-known method used to train an artificial neural network to sort inputs into their respective classes. This is done by adjusting the weights connecting the neurons (input, hidden, and output) of the neural network and the bias weights. A key component of backpropagation is gradient descent which allows for the quickest arrival at a satisfactorily trained network.

The basic process of the algorithm is to feed input data to the network and allow it to proceed through the network as normal. No learning occurs during this stage. Once the output of the network is given by the output neurons, it is compared with target/expected values and an error value is obtained. This error value and gradient values are used in combination with the current weights and outputs of each neuron (after the activation function), and propagated back through the network to adjust each weight of every node in every layer (including the bias weights). This is done for each input in the training data set to complete one epoch. Many epochs may be necessary for the network to learn to a satisfactory point.

2) Implementation

I made use of one-hot encoding for the target values. The number of output neurons are always set to the number of classes with the goal of driving the output for a given output neuron to 0, except in the case that it is the output neuron corresponding to the target class, in which case it should be driven to 1.

Bias values were included with inputs to each neuron. The weight update is done using online learning. For the activation function, I provide the sigmoid logistic function and hyperbolic tangent function.

My implementation is probably best understood by looking through it on a computer, but the pseudocode below, which is mostly taken from the comments I have in the code, may help. For the sake of simplicity, I put all of my code for the project, except for the activation function and activation function derivative, into the same file.

The code for the implementation is organized in this manner:

```
Set Parameters:
      learning rate
      choose activation function
      input data file
      bias used for all neurons
      number of hidden layers
      number of neurons per hidden layer
      number of epochs

Setup input data to be used for neural network
Use one-hot encoding for target values

Initialize weights respecting the fact that the first hidden layer has a
different number of inputs than the remaining layers and the output layer has
a different number of neuron than the rest; pull weight initializations from
uniform distribution over range [-1/sqrt(num_inputs_to_neuron),
1/sqrt(num_inputs_to_neuron)]
```

```
Start Backpropagation Algorithm:
Do for specified number of epochs
      Do for 1 epoch (all inputs once)
            % -------- Do Foward Pass ------------
            Do for first hidden layer
                  Do for each neuron in layer
                        Find v
                        Find y (output of activation function)
                  end
            end
            Do for each remaining hidden layer
                  repeat same procedure with slight variations due to network
      configuration
            end
            Do for output layer
                  repeat same procedure with slight variations due to network
      configuration
            end

            % -------- Do Backward Pass ------------
            Do for each neuron in output layer
                  Find errors for neuron
                  Find local gradient for neuron
                  Do for each neuron in previous (next, if going backwards)
                  layer and bias
                        Find change in weight for current neuron
                  end
            end

            Do for all hidden layers
                  Do for each neuron in the current layer
                        If there is only one hidden layer (it is the first
after the input layer and the last before the output layer)
                              Sum over neurons in output (previous, if going
backwards) layer for use in local gradient calculation
                              Find local gradient for neuron
                              Do for each input element and bias
                              Find change in weight
                        Else if there is more than one hidden layer

                              repeat same procedure with slight variations
      due to network configuration

                        end
                  end
            end
      end

            % -------- Apply Weight Updates ------------
            Add weight vectors and weight update vectors together
      end % End single epoch
end % End training

% -------- Test Network pass code above ------------
for m = 1:num_inputs % Do for all inputs
```

```
      Do for first hidden layer
            Do for each neuron in layer
                  Find v
                  Find y (output of activation function)
            End
      end
      Do for each remaining hidden layer
            repeat same procedure with slight variations due to network
configuration
      end
      Do for output layer
            repeat same procedure with slight variations due to network
configuration
      end
end

Compare results with targets
```

The code for the derivative of the activation function is organized as:

Similarly, the code for the derivative of the activation function is organized as:

3) Experiments and Results

Though the code is designed to work with any of the three data files given (and easily adjusted to suit others), the only one it seems to get close to working correctly for is the threeclouds.data – the one I did the most testing with. For the wine.data file, the network showed very little to no learning regardless of the learning rate. Running semeion.data results in a run time error (I'd fix it, but I think if I stare at this code any longer I will lose what is left of my mind).

The code runs (I think) relatively slowly. Running 5000 epochs on threeclouds.data using 3 hidden layers, 5 neurons per hidden layer, 0.6 for the learning rate, and the sigmoid logistic function as the activation function takes 73 seconds to complete (including time for testing). The results from running on these parameters were:

```
      input 1
      target: 1        0        0
      actual: 0.998679 0.000881 0.000944

      input 2
      target: 1        0        0
      actual: 0.998716 0.000838 0.000961

      ...

      input 99
      target: 1        0        0
      actual: 0.998716 0.000838 0.000961

      input 100
      target: 1        0        0
      actual: 0.998716 0.000838 0.000961
```

```
input 101
target: 0        1        0
actual: 0.000108 0.990581 0.009431

input 102
target: 0        1        0
actual: 0.000096 0.990249 0.011072

...

input 199
target: 0        1        0
actual: 0.000107 0.990653 0.009445

input 200
target: 0        1        0
actual: 0.000107 0.990638 0.009455

input 201
target: 0        0        1
actual: 0.001925 0.000861 0.998643

input 202
target: 0        0        1
actual: 0.001955 0.000895 0.998561

...

input 299
target: 0        0        1
actual: 0.001943 0.000884 0.998590

input 300
target: 0        0        1
actual: 0.001920 0.000857 0.998655
Elapsed time is 73.252630 seconds.
```

Running under the same parameters, but using 2 neurons per layer instead of 5 – to test a less complex network – took almost exactly half the time and produced nearly identical results.

Using the same parameters again, but 3 neurons per layer gave the following output for the first input of the same data set:

```
input 1
target:                    1        0        0
after epoch   20 - actual: 0.070436 0.097649 0.876872
after epoch   40 - actual: 0.907696 0.005571 0.107805
after epoch   60 - actual: 0.964835 0.003630 0.037154
after epoch   80 - actual: 0.975792 0.003082 0.024057
after epoch  100 - actual: 0.981129 0.002878 0.018326
after epoch  120 - actual: 0.983946 0.002736 0.015658
after epoch  140 - actual: 0.985779 0.002635 0.013858
after epoch  160 - actual: 0.987101 0.002557 0.012545
after epoch  180 - actual: 0.988116 0.002495 0.011535
```

```
after epoch  200 - actual: 0.988927 0.002441 0.010727
after epoch  220 - actual: 0.989596 0.002395 0.010062
after epoch  240 - actual: 0.990160 0.002355 0.009502
after epoch  260 - actual: 0.990643 0.002318 0.009022
after epoch  280 - actual: 0.991063 0.002285 0.008604
after epoch  300 - actual: 0.991432 0.002254 0.008237
after epoch  320 - actual: 0.991760 0.002226 0.007910
after epoch  340 - actual: 0.992055 0.002200 0.007616
after epoch  360 - actual: 0.992320 0.002175 0.007351
after epoch  380 - actual: 0.992561 0.002152 0.007110
after epoch  400 - actual: 0.992782 0.002131 0.006889
after epoch  420 - actual: 0.992984 0.002111 0.006687
after epoch  440 - actual: 0.993171 0.002092 0.006499
after epoch  460 - actual: 0.993343 0.002074 0.006326
after epoch  480 - actual: 0.993504 0.002057 0.006164
after epoch  500 - actual: 0.993653 0.002042 0.006013
```

Using the same parameters but with a learning rate of 0.2 instead of 0.6 gave:

```
input 1
target:                    1        0        0
after epoch   20 - actual: 0.246012 0.257587 0.298045
after epoch   40 - actual: 0.787602 0.432058 0.002107
after epoch   60 - actual: 0.944664 0.067983 0.000757
after epoch   80 - actual: 0.962424 0.038172 0.000569
after epoch  100 - actual: 0.971383 0.029165 0.000478
after epoch  120 - actual: 0.975787 0.025018 0.000425
after epoch  140 - actual: 0.978127 0.022288 0.000390
after epoch  160 - actual: 0.979726 0.020211 0.000365
after epoch  180 - actual: 0.980929 0.018560 0.000345
after epoch  200 - actual: 0.981872 0.017212 0.000330
after epoch  220 - actual: 0.982630 0.016081 0.000318
after epoch  240 - actual: 0.983253 0.015108 0.000310
after epoch  260 - actual: 0.983772 0.014256 0.000304
after epoch  280 - actual: 0.984212 0.013499 0.000301
after epoch  300 - actual: 0.984591 0.012823 0.000300
after epoch  320 - actual: 0.984924 0.012219 0.000300
after epoch  340 - actual: 0.985220 0.011683 0.000302
after epoch  360 - actual: 0.985483 0.011220 0.000304
after epoch  380 - actual: 0.985691 0.010891 0.000304
after epoch  400 - actual: 0.985882 0.010563 0.000309
after epoch  420 - actual: 0.986267 0.010359 0.000308
after epoch  440 - actual: 0.986647 0.009963 0.000314
after epoch  460 - actual: 0.987122 0.009671 0.000315
after epoch  480 - actual: 0.987585 0.009453 0.000317
after epoch  500 - actual: 0.987976 0.009201 0.000321
```

Using the same parameters again but with a learning rate of 0.95 gave:

```
input 1
target:                    1        0        0
after epoch   20 - actual: 0.040946 0.069322 0.916726
after epoch   40 - actual: 0.018871 0.056263 0.929161
after epoch   60 - actual: 0.586064 0.097767 0.076267
after epoch   80 - actual: 0.917588 0.018129 0.028098
```

```
after epoch  100 - actual: 0.986011 0.011427 0.009502
after epoch  120 - actual: 0.989085 0.009873 0.008700
after epoch  140 - actual: 0.992551 0.008819 0.007163
after epoch  160 - actual: 0.992820 0.008144 0.007780
after epoch  180 - actual: 0.992730 0.007524 0.008538
after epoch  200 - actual: 0.992628 0.007363 0.008338
after epoch  220 - actual: 0.993819 0.007132 0.007443
after epoch  240 - actual: 0.994075 0.006762 0.007284
after epoch  260 - actual: 0.994452 0.007289 0.006212
after epoch  280 - actual: 0.993953 0.006601 0.007837
after epoch  300 - actual: 0.994141 0.006103 0.007422
after epoch  320 - actual: 0.994404 0.006306 0.006987
after epoch  340 - actual: 0.994609 0.006599 0.006787
after epoch  360 - actual: 0.994786 0.006763 0.006732
after epoch  380 - actual: 0.994951 0.006746 0.006754
after epoch  400 - actual: 0.995109 0.006592 0.006787
after epoch  420 - actual: 0.995254 0.006367 0.006794
after epoch  440 - actual: 0.995383 0.006122 0.006768
after epoch  460 - actual: 0.995492 0.005887 0.006720
after epoch  480 - actual: 0.995580 0.005676 0.006660
after epoch  500 - actual: 0.995645 0.005501 0.006593
```

I additionally ran a few trials keeping all parameters but the activation function the same. For the sigmoid logistic function, where the desired output is 1, the result is at about 0.997 after epoch 1000. Using the hyperbolic tangent function, however, the value seems to consistently be around 0.987 after epoch 1000, but in earlier epochs, quicker learning is seen.

4) Discussion

Running my implementation on the neural network on the threeclouds.data file seem to be quite successful. It is clear that the "largest amount" of learning happens within the first several epochs, while the following hundreds/thousands of epochs make only small adjustments to slowly reach the target values. Changing the complexity of the network did not seem to alter the results in any major way. I was surprised that increasing the learning rate to 0.95 from 0.6 appeared to slow down the learning (I don't think it overshot the target here). It would be interesting to see if this held for several trials. It would also be interesting to see how more or less input data for a single epoch changes the learning rate per epoch. It is obvious that the activation function makes an impact on the results, though it seems that the best function would be found through trial and error.

Other than getting the code to work for the other data sets, there are a number of improvements that could be made to my implementation. These include:

- I could have made more use of Matlab's matrix multiplication abilities and used fewer nested for loops.
- Regarding the long running time, I think there is some excellent opportunity for parallelization in running neural networks.
- Randomizing the order that the inputs are presented to the network in each epoch may help avoid overfitting.

5) Program Code

See attached.

```
% --------------------------------------------------------------------------
% Thomas Nabelek
% September 22, 2016

% ECE 7870
% Project 1 - Backpropagation training and testing code
% --------------------------------------------------------------------------

clear;
tic;

% Parameters ------------------------------------------------------------
learning_rate = 0.5;     % Range 0 (no learning) to 1
activation_func_opt = 2;     % Set to 1 for sigmoid (logistic) function or to 2 for hyperbolic tangent function
data_file = 'threeclouds.data';
bias = 1;
num_hidden_layers = 3;
num_neurons_per_layer = 3;
num_epochs = 1000;
% -----------------------------------------------------------------------

% -------------------- Setup data --------------------
data_raw = load(data_file);
num_layers = num_hidden_layers + 1;  % Add 1 for output layer

if (strcmp(data_file, 'threeclouds.data') || strcmp(data_file, 'wine.data'))
    if (strcmp(data_file, 'threeclouds.data'))
        num_classes = 3;
        data = data_raw(:,2:3);
    else
        num_classes = 3;
        data = data_raw(:,2:14);
    end
    num_inputs = size(data, 1);
    input_dimension = size(data, 2);

    % Do one-hot encoding for target values
    targets = zeros(num_inputs, num_classes);
    for n = 1:num_inputs
        class = data_raw(n, 1);
        targets(n, class) = 1;
    end

else % For semeion.data file
    num_classes = 10;
    data = data_raw(:,1:256);
    num_inputs = size(data, 1);
    input_dimension = size(data, 2);
    targets = data_raw(:,257:266);
end

num_output_neurons = num_classes;
data(:, input_dimension+1) = 1;  % Assign bias of 1 for every input vector

% ---------- Weight initialization - (first hidden layer has different number of inputs than remaining layers and output has different number of neurons)
num_inputs_to_neuron = input_dimension;
```

```
57 weights_first_hidden_layer = (rand([num_inputs_to_neuron+1 num_neurons_per_layer 1]) - 0.5) * 2 * (1/sqrt(num_inputs_to_neuron));  % Initialize weights to range [-1/sqrt(num_inputs_to_neuron), 1/sqrt(num_inputs_to_neuron)]; each neuron has its own column of weights
58 num_inputs_to_neuron = num_neurons_per_layer;
59 weights = (rand([num_inputs_to_neuron+1 num_neurons_per_layer num_layers-2]) - 0.5) * 2 * (1/sqrt(num_inputs_to_neuron));  % Initialize weights to range [-1/sqrt(num_inputs_to_neuron), 1/sqrt(num_inputs_to_neuron)]; each neuron has its own column of weights
60 weights_output_layer = (rand([num_inputs_to_neuron+1 num_output_neurons 1]) - 0.5) * 2 * (1/sqrt(num_inputs_to_neuron));  % Initialize weights to range [-1/sqrt(num_inputs_to_neuron), 1/sqrt(num_inputs_to_neuron)]; each neuron has its own column of weights
61
62 % ---------- Start Backpropagation Algorithm (1 epoch) ----------------
63 % disp(sprintf('\ninput %d\ntarget: %20d %8d %8d', 1, targets(1, 1), targets(1, 2), targets(1, 3)));
64 for t = 1:num_epochs  % Do for specified number of epochs
65     for m = 1:num_inputs  % Do for 1 epoch (all inputs once)
66
67         % Store all v, y, and local_gradient values for a given layer in the same column in their respective arrays
68
69         % -------- Do Foward Pass ------------
70         n = 1;  % Do for first hidden layer where num_inputs_to_neuron = input_dimension
71         for h = 1:num_neurons_per_layer  % Do for each neuron in layer
72             v(h, n) = weights_first_hidden_layer(:, h, n)'*data(m, :)';   % Multiply each input to neuron with appropriate weight, and sum to find v for current neuron
73             y(h, n) = activation_function(v(h, n), activation_func_opt);   % Get activation function output for current neuron
74         end
75         y(h+1, n) = 1;  % Add 1 for the bias of the next neuron
76
77         for n = 2:num_layers-1  % Do for each remaining hidden layer (subtact 1 for output layer)
78             for h = 1:num_neurons_per_layer  % Do for each neuron in layer
79                 v(h, n) = weights(:, h, n-1)'*y(:, n-1);   % Multiply each input to neuron with appropriate weight, and sum to find v for current neuron
80                 y(h, n) = activation_function(v(h, n), activation_func_opt);   % Get activation function output for current neuron
81             end
82             y(h+1, n) = 1;  % Add 1 for the bias of next neuron
83         end
84
85         n = num_layers;  % Move to output layer
86         for h = 1:num_output_neurons  % Do for each neuron in output layer
87             v(h, n) = weights_output_layer(:, h)'*y(:, n-1);   % Multiply each input to neuron with appropriate weight, and sum to find v for current neuron
88             y(h, n) = activation_function(v(h, n), activation_func_opt);   % Get activation function output for current neuron
89         end
90
91         % -------- Do Backward Pass ------------
92         for h = 1:num_output_neurons  % Do for each neuron in output layer
93             error(h) = targets(m, h) - y(h, n);   %#ok<*SAGROW> % Find errors for output neurons
94             local_gradient(h, n) = error(h) * activation_deriv_function(v(h, n), activation_func_opt);   % Find local gradient for neuron
95             for l = 1:num_neurons_per_layer+1  % Do for each neuron in previous (next, if going backwards) layer and bias
96                 w_delta_output(l, h) = learning_rate * local_gradient(h, n) * y(l, n-1);
97             end
98         end
99
```

```
100         for n = n-1:-1:1  % Do for all hidden layers
101             for h = 1:num_neurons_per_layer  % Do for each neuron in the current layer
102                 sum = 0;
103                 if (num_hidden_layers == 1)  % If there is only one hidden layer (it is the first after the input layer and the last before the output layer)
104                         for l = 1:num_output_neurons  % Sum over neurons in output (previous, if going backwards) layer
105                             sum = sum + local_gradient(l, n+1)*weights_output_layer(h, l);    % For weights, n+1-1 = n; subtract 1 because weights for first hidden layer are in separate array (weights(x,x,n) are next layer's weights)
106                         end
107                         local_gradient(h, n) = activation_deriv_function(v(h, n), activation_func_opt) * sum; % Find local gradient for neuron
108                         for l = 1:input_dimension+1  % Do for each input element and bias
109                             w_delta_first_hidden_layer(l, h) =  learning_rate * local_gradient(h, n) * data(m, l);
110                         end
111
112                 else %If there is more than one hidden layer
113                     if (n == num_layers-1)  % If current layer is last hidden layer before output layer
114                         for l = 1:num_output_neurons  % Sum over neurons in output layer
115                             sum = sum + local_gradient(l, n+1)*weights_output_layer(h, l);
116                         end
117                         local_gradient(h, n) = activation_deriv_function(v(h, n), activation_func_opt) * sum; % Find local gradient for neuron
118                         for l = 1:num_neurons_per_layer+1  % Do for each neuron in previous (next, if going backwards) layer and bias
119                             w_delta(l, h, n-1) =  learning_rate * local_gradient(h, n) * y(l, n-1);
120                         end
121
122                     elseif (n > 1) % If current layer is not first or last hidden layer
123                         for l = 1:num_neurons_per_layer  % Sum over neurons in next (previous, if going backwards) layer
124                             sum = sum + local_gradient(l, n+1)*weights(h, l, n);    % For weights, n+1-1 = n; subtract 1 because weights for first hidden layer are in separate array (weights(x,x,n) are next layer's weights)
125                         end
126                         local_gradient(h, n) = activation_deriv_function(v(h, n), activation_func_opt) * sum; % Find local gradient for neuron
127                         for l = 1:num_neurons_per_layer+1  % Do for each neuron in previous (next, if going backwards) layer and bias
128                             w_delta(l, h, n-1) =  learning_rate * local_gradient(h, n) * y(l, n-1);
129                         end
130
131                     else % If current layer is first hidden layer
132                         for l = 1:num_neurons_per_layer  % Sum over neurons in next (previous, if going backwards) layer
133                             sum = sum + local_gradient(l, n+1)*weights(h, l, n);    % For weights, n+1-1 = n; subtract 1 because weights for first hidden layer are in separate array (weights(x,x,n) are next layer's weights)
134                         end
135                         local_gradient(h, n) = activation_deriv_function(v(h, n), activation_func_opt) * sum; % Find local gradient for neuron
136                         for l = 1:input_dimension+1  % Do for each input element and bias
137                             w_delta_first_hidden_layer(l, h) =  learning_rate * local_gradient(h, n) * data(m, l);
138                         end
139
```

```
                    end
                end
            end
        end
        if (mod(t, 20) == 0 && m == 1)
            disp(sprintf( 'after epoch %4d - actual: %f %f %f' , t, y(1, num_layers), y(2, num_layers), y(3, num_layers)));
        end

        % -------- Apply Weight Updates ------------
        weights_first_hidden_layer = weights_first_hidden_layer + w_delta_first_hidden_layer;
        weights_output_layer = weights_output_layer + w_delta_output;
        weights = weights + w_delta;

    end % End single epoch
%     if (mod(t, 500) == 0)
%         disp(sprintf('epoch %5d completed', t));
%     end

end % End training

toc;

% -------- Test Network (required for project) - mostly the same as forward pass code above ------------
for m = 1:num_inputs  % Do for all inputs

    % Store all v and y values for a given layer in the same column in their respective arrays

    n = 1;  % Do for first hidden layer where num_inputs_to_neuron = input_dimension
    for h = 1:num_neurons_per_layer  % Do for each neuron in layer
         v(h, n) = weights_first_hidden_layer(:, h, n)'*data(m, :)';   % Multiply each input to neuron with appropriate weight, and sum to find v for current neuron
         y(h, n) = activation_function(v(h, n), activation_func_opt);   % Get activation function output for current neuron
    end
    y(h+1, n) = 1;  % Add 1 for the bias of the next neuron

    for n = 2:num_layers-1  % Do for each remaining hidden layer (subtact 1 for output layer)
        for h = 1:num_neurons_per_layer  % Do for each neuron in layer
            v(h, n) = weights(:, h, n-1)'*y(:, n-1);   % Multiply each input to neuron with appropriate weight, and sum to find v for current neuron
            y(h, n) = activation_function(v(h, n), activation_func_opt);   % Get activation function output for current neuron
        end
        y(h+1, n) = 1;  % Add 1 for the bias of next neuron
    end

    n = num_layers;  % Move to output layer
    for h = 1:num_output_neurons  % Do for each neuron in output layer
        v(h, n) = weights_output_layer(:, h)'*y(:, n-1);   % Multiply each input to neuron with appropriate weight, and sum to find v for current neuron
        y(h, n) = activation_function(v(h, n), activation_func_opt);   % Get activation function output for current neuron
    end

%     disp(sprintf('\ninput %d\ntarget: %d %8d %8d\nactual: %f %f %f', m, targets(m, 1), targets(m, 2), targets(m, 3), y(1, num_layers), y(2, num_layers), y(3, num_layers)));
```

```
189
190 end
```

```
function y = activation_function(v, opt)
    if (opt == 1) % Sigmoid (logistic) function
        a = 1; % Slope paramete - if changed, change in derivative function too
        y = 1 / (1 + exp(-a*v));
    end
    if (opt == 2) % Hyperbolic tangent function
        a = 1; % If changed, change in derivative function too
        b = 1; % If changed, change in derivative function too
        y = a*tanh(b*v);
    end
end
```

```
function y = activation_deriv_function(v, opt)
    if (opt == 1) % Sigmoid (logistic) function
        a = 1; % Slope parameter - if changed, change in original function too
        y = (a*exp(-a*v)) / (1 + exp(-a*v))^2;
    end
    if (opt == 2) % Hyperbolic tangent function
        a = 1; % If changed, change in original function too
        b = 1; % If changed, change in original function too
        y = a*b*sech(b*v)^2;
    end
end
```