Pseudocode

```
Initialize a cell for weights
Initialize a cell for biases

For each layer starting from the second layer
    Initialize weights with normally distributed values and fill in its corresponding position in
    the weight cell
    Initialize biases with normally distributed values and fill in its corresponding position in
    the biases cell

Initialize a cell to store batch values
Initialize a cell to store target values
Initialize a counter to keep tracking batch indexes

For each starting index of the instance of each batch
    If desired batch size is greater than the actual batch size
        Insert the batch with actual size into the batch cell
        Insert the corresponding target batch into the target cell
    Else
        Insert the batch with desired size into the batch cell
        Insert the corresponding target batch into the target cell
        Increment the batch index counter by one

For each epoch
    For each batch
        Initialize a cell to store intermediate values
        Initialize a cell to store activations
        Set the first element of activation cell to the input batch

        For each layer starting from the second layer
            Intermediate value is calculated
            Activation values is calculated by transforming the intermediate value
            which is then stored into the activation cell

        Calculate error which is the difference between activations and targets
        Calculate sigmoid gradient with respect to the intermediate values
        Calculate the cost by multiply the error and sigmoid gradient
        Initiate a cell for deltas
        Set the last element of the delta cell to the cost

        For each layer starting from the second to the last layer to the second layer
            Delta of that layer is calculated by multiplying weights with deltas and
            multiplying the product with sigmoid gradient
```

```
                For each layer starting from the last layer to the second layer
                    Update the weights by subtracting learning rate divided by the total
                    number of observations in the batch times the result of delta of the layer
                    multiplies with the next layer's activations, from current weights
                    Update the biases by subtracting learning rate divided by the total
                    number of observations in the batch and then multiplying by the sum of
                    the deltas in the layer, from the current biases

        Initialize the final output cell
        Set the first element of the output cell to inputs

        For each layer starting from the second layer to the end
            Caculate intermediate values
            Store transformed intermediate values into the output cell

        Calculate the mean squared of error which is the squared errors divided by two times
        the total observations

        If it is a multiclass problem
            Get the matrix location of the max value of the final output
            Get the matrix location of the max value of the targets
            Subtract the two location vectors
            Count the number of zeros in the vector which are the correct cases
        Else
            Round up or down the final output vecotr
            Count the number of zeros in the vector which are the correct cases

        Calculate the overall accuracy

        If the number of input epoch is less or equal to 100
            Display iteration number, MSE, correct cases over total cases and accuracy
        Else if the number of input epoch is greater or equal to 100 and the MOD of epoch and
        100 and equal to 0
            Display iteration number, MSE, correct cases over total cases and accuracy of
            that epoch

        If the accuracy is 1
            Display iteration number, MSE, correct cases over total cases and accuracy of
            that epoch
            Stop the function

If number of epoch is greater than 100 and the MOD of number of total epochs and 100 does
not equal to
        Display iteration number, MSE, correct cases over total cases and accuracy of that epoch
```

Instruction

My neural network code is stored in the file MyNetwork.m and the function name is called MyNetwork. The required input parameters are as following,

- inputs: a matrix with a column for each example, and a row for each input feature
- targets: a matrix with a column for each example, and a row for each output feature
- nodeLayers: a vector with the number of nodes in each layer (including the input and output layers).
- numEpochs: (scalar) desired number of epochs to run
- batchSize: (scalar) number of instances in a mini-batch
- eta: (scalar) learning rate

The output of my network are the optimized weight and bias matrices. In order to store this value of these matrices, you should run the code such as following,

[matrix, bias] = MyNetwork(*input parameters*);

Here is an example of running of my code with parameters to solve the XOR problem,

```
>> input = [1 1 0 0;  1 0 1 0];
>> target = [0 1 1 0];
>> layer = [2 3 1];
>> epoch = 40;
>> batch = 10
>> eta = 0.01;
>> [matrix, bias] = MyNetwork(input, target, layer, epoch, batch, eta);
```

Description

MyNetwork is a conventional neural network application implemented in Matlab. This system uses the basic learning and error propagation algorithms with mini batch approach to train the network. Here is an overview of how the application works:

Step One: Weight and Biases Initialization

Because weights and biases are what the neural network is learning on, we need to store them some and initialize them with normally distributed values, meaning with the mean of zero and standard deviation of one. In the Pseudocode, it is shown between line 2 and 9.

Step Two: Batching

Batching is an ensemble learning method for the network, meaning we divide the training data into batches and learn the weight and biases based on each batch. It can also be perceived as fine tune the network with many small size samples. In the Pseudocode, it is indicated between line 11 and 22.

Step Three: Start Learning Epoch by Epoch and Batch by Batch (Backpropagation and Gradient Descent)

This is the step where the network learns to adjust its weight and biases through iteration and iteration. For each epoch, the network will learn to adjust its weights and biases by the number of batch times. For example, if there are ten batches in the training set, the network will learn to adjust its weights and biases ten times per epoch. The learning process is achieved first, by back propagating the errors in the network, per batch time, which is shown in the Pseudocode

between line 41 and 43, then second, by gradient descent algorithms, which is shown in the Pseudocode between line 45 and 51.

Step Four: Calculate the Final Predicted Value and Evaluation

After learning the weights and biases iterations through iterations, the final set of updated weights and biases are used to predict the final output using the entire input training example. Then the number of correctly classified cases will be calculated and the overall accuracy will be calculated as well. This part of the application is shown in the Pseudocode between line 53 and 88.

Code

```
% author: Taihua(Ray) Li
% date: Oct 9, 2016
function [ weight, bias ] = MyNetwork(inputs, targets, nodeLayers, numEpochs,
batchSize, eta)
% this is the project one for CSC 578, Neural Network and Deep Learning
% step one: initilize weights and bias matrices
weight = {}; % initialize weigt matrix with cell
bias = {}; % initialize bias matrix with cell
for toLayer = 2:length(nodeLayers) % for each nodeLayer input
    weight{toLayer} =normrnd(0, 1, nodeLayers(toLayer), nodeLayers(toLayer-
1)); % initialize the weights
    bias{toLayer} = normrnd(0, 1, nodeLayers(toLayer), 1); % initialize the
bias
    % initialize weights and biases with normal distribution, mean = 0, sd=1
end

% step two: batching
totNumInst = size(inputs, 2); % total number of samples
targetbatches = {}; % to store target batches
batches = {}; % to store batches
numbatch = 1; % to index the position of each batch
for StartInsPos = 1:batchSize:totNumInst % divide the sample into minibatches
    if totNumInst-StartInsPos < batchSize % this avoids index out of range
problem
        batch = inputs(:, StartInsPos:end); % a matrix representing the batch
        batches{numbatch} = batch;
        targetbatches{numbatch} = targets(:, StartInsPos:end);
    else
        batch = inputs(:, StartInsPos:StartInsPos+batchSize-1); % a matrix
representing the batch
        batches{numbatch} = batch;
        targetbatches{numbatch} = targets(:, StartInsPos+batchSize-1);
        numbatch = numbatch + 1 ;
    end
end

% step three: start epochs
for iteration = 1:numEpochs % for each epochs
    for batchindex = 1:numbatch % for each batch
        z = {}; % to store intermediate values
        activations = {}; % initilize the activation matrix
        activations{1} = batches{batchindex}; % first activation layer is the
input layer
        deltas = {}; % to store deltas for gradient descent
        for layer = 2:length(nodeLayers)
```

```
            z{layer} = bsxfun(@plus, (weight{layer} * activations{layer-1}),
bias{layer});
            activations{layer} = logsig(z{layer}); % calculate and store the
activation for next layer
        end
        % calculate output error aka cost
        % criterion: quadratic cost function
        error = activations{length(nodeLayers)} - targetbatches{batchindex};
        sigprime = logsig(z{length(nodeLayers)}).*(1-
logsig(z{length(nodeLayers)}));
        cost = error.*sigprime; % this is the delta of last layer

        % step four: backpropagation
        deltas{length(nodeLayers)} = cost; % last layer of delta is the
quadratic cost function output
        for layer = (length(nodeLayers)-1):-1:2 % skip the last layer
            deltas{layer} =
(weight{layer+1}.'*deltas{layer+1}).*(logsig(z{layer}).*(1-
logsig(z{layer})));
        end

        % step five: gradient descent
        for layer = length(nodeLayers):-1:2 % start from the last layer
            w = weight{layer} -
eta/length(batches{batchindex})*deltas{layer}*activations{layer-1}.';
            weight{layer} = w;
            b = bias{layer} -
eta/length(batches{batchindex})*sum(deltas{layer}, 2);
            bias{layer} = b;
        end
    end

    % step six: calculate output statistics and display a message for each
epoch
    output = {}; % to store the output calculated using updated weights
    output{1} = inputs;
    for layer = 2:length(nodeLayers)
        z2 = bsxfun(@plus,(weight{layer} * output{layer-1}), bias{layer});
        output{layer} = logsig(z2); % calculate and store the activation for
next layer
    end

    % calculate MSE, number of correct cases and accuracy rate
    error = output{length(nodeLayers)}-targets;
    MSE = sqrt(sum(sum(error.^2)))/(2*length(inputs)); % mean() only divide
by n, but we want to divde by 2n

    if size(targets, 1) > 1 % if it is a multi class problem
        [mx, loc] = max(output{length(nodeLayers)}); % max value of each
column/observation
        [mx2, loct] = max(targets);
        t = loct - loc;
        correct = sum(t(:)==0);
    else % if it is a binary classification problem, we set the threshold at
0.5
        positive = round(output{length(nodeLayers)});
        correct = sum(abs(targets-positive)==0); % incorrect cases will have
```

```
a sum greater than zero
    end

    totNumCase = size(inputs, 2);
    accuracy = correct/totNumCase;

    if numEpochs <= 100 % print message for each iteration if running a small
number of epochs
        fprintf('Epoch %d, MSE: %f, Correct: %d/%d, Acc: %f \n', iteration,
MSE, correct, totNumCase, accuracy);
    elseif mod(iteration, 100) == 0 && numEpochs > 100
        % print only hundredth iteration's message if running a big number of
epochs
        fprintf('Epoch %d, MSE: %f, Correct: %d/%d, Acc: %f \n', iteration,
MSE, correct, totNumCase, accuracy);
    end

    % another stopping criterion: accuracy = 1
    if correct == totNumCase
        fprintf('Epoch %d, MSE: %f, Correct: %d/%d, Acc: %f \n', iteration,
MSE, correct, totNumCase, accuracy);
        break
    end
end

if mod(numEpochs, 100) ~= 0 && numEpochs > 100
    % print the result for the last iteration for large
    fprintf('Epoch %d, MSE: %f, Correct: %d/%d, Acc: %f \n', numEpochs, MSE,
correct, totNumCase, accuracy);
end

end
```

Analysis

My application works properly for all dataset with multiple configurations, including changes in number of hidden layer, number of epochs, learning rate (eta) and batch size. The tested output from different dataset can be found later in this report. To evaluate the efficiency of my application, I manually changed some parameters to run on the XOR problem. Here is the result:

Attempt 1

Configurations:

Epochs: 10 Hidden nodes: 2 Batchsize: 4 eta/learning rate: 0.1

Accuracy: 0.5

Attempt 2

Configurations:

Epochs: 1,000 Hidden nodes: 2 Batchsize: 4 eta/learning rate: 0.1

Accuracy: 1 at epoch 284

Attempt 3

Configurations:

Epochs: 1,000,000 Hidden nodes: 2 Batchsize: 4 eta/learning rate: 0.1

Accuracy: 0.5

With different attempts of solving XOR problem by only changing epoch parameter, it shows that the current application is really unstable as It could converge to the ideal result with a low number of iterations and sometimes it might not even converge because the neurons are saturated. Overall, this application is really unstable due to the usage of sigmoid transformation function and to improve the algorithm, some ideas are mentioned in the next section.

Ideas for Enhancement

1. Replace quadratic cost function and sigmoid transformation function with cross entropy cost function and softmax transformation function if the task is a multiclass problem
2. Implement training, testing and validation sets to avoid under- or overfitting problems
3. Auto training. Without set the number of epoch, the network will learn until it converges or stop before overfitting
4. Better weight and biases initialization methods to avoid saturated neurons

Outputs

I used three popularly used datasets to test my application with different parameter configurations. Here are the result indicating MSE and accuracy of each epoch for each dataset with each configuration:

Iris Dataset

Configurations:

Epochs: 100 Hidden nodes: 20 Batchsize: 10 eta/learning rate: 0.1

Matlab Code:

```
>> iris = csvread('iris.csv');
>> inputs = iris(:, 1:4).';
>> targets = iris(:, 5:7).';
>> MyNetwork(inputs, targets, [4, 20, 3], 100, 10, 0.1);
```

Matlab output:

```
Epoch 1, MSE: 0.239807, Correct: 0/150, Acc: 0.000000
Epoch 2, MSE: 0.214267, Correct: 50/150, Acc: 0.333333
Epoch 3, MSE: 0.193662, Correct: 50/150, Acc: 0.333333
Epoch 4, MSE: 0.180164, Correct: 50/150, Acc: 0.333333
Epoch 5, MSE: 0.172454, Correct: 50/150, Acc: 0.333333
Epoch 6, MSE: 0.164808, Correct: 50/150, Acc: 0.333333
Epoch 7, MSE: 0.151399, Correct: 50/150, Acc: 0.333333
Epoch 8, MSE: 0.122116, Correct: 56/150, Acc: 0.373333
Epoch 9, MSE: 0.109902, Correct: 94/150, Acc: 0.626667
Epoch 10, MSE: 0.103680, Correct: 59/150, Acc: 0.393333
Epoch 11, MSE: 0.099211, Correct: 60/150, Acc: 0.400000
Epoch 12, MSE: 0.095724, Correct: 82/150, Acc: 0.546667
Epoch 13, MSE: 0.092804, Correct: 99/150, Acc: 0.660000
Epoch 14, MSE: 0.090139, Correct: 100/150, Acc: 0.666667
Epoch 15, MSE: 0.087529, Correct: 100/150, Acc: 0.666667
Epoch 16, MSE: 0.084818, Correct: 100/150, Acc: 0.666667
```

Epoch 17, MSE: 0.081878, Correct: 100/150, Acc: 0.666667
Epoch 18, MSE: 0.078693, Correct: 100/150, Acc: 0.666667
Epoch 19, MSE: 0.075529, Correct: 100/150, Acc: 0.666667
Epoch 20, MSE: 0.072772, Correct: 100/150, Acc: 0.666667
Epoch 21, MSE: 0.070549, Correct: 101/150, Acc: 0.673333
Epoch 22, MSE: 0.068762, Correct: 101/150, Acc: 0.673333
Epoch 23, MSE: 0.067274, Correct: 102/150, Acc: 0.680000
Epoch 24, MSE: 0.065990, Correct: 102/150, Acc: 0.680000
Epoch 25, MSE: 0.064850, Correct: 103/150, Acc: 0.686667
Epoch 26, MSE: 0.063819, Correct: 103/150, Acc: 0.686667
Epoch 27, MSE: 0.062873, Correct: 104/150, Acc: 0.693333
Epoch 28, MSE: 0.061997, Correct: 104/150, Acc: 0.693333
Epoch 29, MSE: 0.061180, Correct: 104/150, Acc: 0.693333
Epoch 30, MSE: 0.060414, Correct: 104/150, Acc: 0.693333
Epoch 31, MSE: 0.059691, Correct: 106/150, Acc: 0.706667
Epoch 32, MSE: 0.059008, Correct: 106/150, Acc: 0.706667
Epoch 33, MSE: 0.058360, Correct: 106/150, Acc: 0.706667
Epoch 34, MSE: 0.057742, Correct: 107/150, Acc: 0.713333
Epoch 35, MSE: 0.057152, Correct: 107/150, Acc: 0.713333
Epoch 36, MSE: 0.056587, Correct: 109/150, Acc: 0.726667
Epoch 37, MSE: 0.056045, Correct: 111/150, Acc: 0.740000
Epoch 38, MSE: 0.055522, Correct: 111/150, Acc: 0.740000
Epoch 39, MSE: 0.055018, Correct: 111/150, Acc: 0.740000
Epoch 40, MSE: 0.054530, Correct: 111/150, Acc: 0.740000
Epoch 41, MSE: 0.054058, Correct: 112/150, Acc: 0.746667
Epoch 42, MSE: 0.053600, Correct: 112/150, Acc: 0.746667
Epoch 43, MSE: 0.053154, Correct: 112/150, Acc: 0.746667
Epoch 44, MSE: 0.052719, Correct: 112/150, Acc: 0.746667
Epoch 45, MSE: 0.052295, Correct: 114/150, Acc: 0.760000
Epoch 46, MSE: 0.051881, Correct: 115/150, Acc: 0.766667
Epoch 47, MSE: 0.051476, Correct: 115/150, Acc: 0.766667
Epoch 48, MSE: 0.051080, Correct: 115/150, Acc: 0.766667
Epoch 49, MSE: 0.050691, Correct: 115/150, Acc: 0.766667
Epoch 50, MSE: 0.050309, Correct: 115/150, Acc: 0.766667
Epoch 51, MSE: 0.049934, Correct: 115/150, Acc: 0.766667
Epoch 52, MSE: 0.049566, Correct: 116/150, Acc: 0.773333
Epoch 53, MSE: 0.049204, Correct: 118/150, Acc: 0.786667
Epoch 54, MSE: 0.048848, Correct: 118/150, Acc: 0.786667
Epoch 55, MSE: 0.048497, Correct: 118/150, Acc: 0.786667
Epoch 56, MSE: 0.048152, Correct: 120/150, Acc: 0.800000
Epoch 57, MSE: 0.047812, Correct: 120/150, Acc: 0.800000
Epoch 58, MSE: 0.047477, Correct: 120/150, Acc: 0.800000
Epoch 59, MSE: 0.047147, Correct: 123/150, Acc: 0.820000
Epoch 60, MSE: 0.046822, Correct: 124/150, Acc: 0.826667

Epoch 61, MSE: 0.046502, Correct: 124/150, Acc: 0.826667
Epoch 62, MSE: 0.046186, Correct: 125/150, Acc: 0.833333
Epoch 63, MSE: 0.045875, Correct: 125/150, Acc: 0.833333
Epoch 64, MSE: 0.045568, Correct: 125/150, Acc: 0.833333
Epoch 65, MSE: 0.045266, Correct: 126/150, Acc: 0.840000
Epoch 66, MSE: 0.044967, Correct: 126/150, Acc: 0.840000
Epoch 67, MSE: 0.044673, Correct: 126/150, Acc: 0.840000
Epoch 68, MSE: 0.044383, Correct: 126/150, Acc: 0.840000
Epoch 69, MSE: 0.044096, Correct: 126/150, Acc: 0.840000
Epoch 70, MSE: 0.043813, Correct: 126/150, Acc: 0.840000
Epoch 71, MSE: 0.043534, Correct: 127/150, Acc: 0.846667
Epoch 72, MSE: 0.043258, Correct: 127/150, Acc: 0.846667
Epoch 73, MSE: 0.042986, Correct: 127/150, Acc: 0.846667
Epoch 74, MSE: 0.042716, Correct: 127/150, Acc: 0.846667
Epoch 75, MSE: 0.042450, Correct: 128/150, Acc: 0.853333
Epoch 76, MSE: 0.042186, Correct: 128/150, Acc: 0.853333
Epoch 77, MSE: 0.041926, Correct: 128/150, Acc: 0.853333
Epoch 78, MSE: 0.041668, Correct: 128/150, Acc: 0.853333
Epoch 79, MSE: 0.041412, Correct: 128/150, Acc: 0.853333
Epoch 80, MSE: 0.041159, Correct: 128/150, Acc: 0.853333
Epoch 81, MSE: 0.040908, Correct: 128/150, Acc: 0.853333
Epoch 82, MSE: 0.040660, Correct: 128/150, Acc: 0.853333
Epoch 83, MSE: 0.040414, Correct: 128/150, Acc: 0.853333
Epoch 84, MSE: 0.040170, Correct: 128/150, Acc: 0.853333
Epoch 85, MSE: 0.039927, Correct: 128/150, Acc: 0.853333
Epoch 86, MSE: 0.039687, Correct: 128/150, Acc: 0.853333
Epoch 87, MSE: 0.039449, Correct: 129/150, Acc: 0.860000
Epoch 88, MSE: 0.039212, Correct: 129/150, Acc: 0.860000
Epoch 89, MSE: 0.038978, Correct: 129/150, Acc: 0.860000
Epoch 90, MSE: 0.038744, Correct: 129/150, Acc: 0.860000
Epoch 91, MSE: 0.038513, Correct: 129/150, Acc: 0.860000
Epoch 92, MSE: 0.038283, Correct: 129/150, Acc: 0.860000
Epoch 93, MSE: 0.038055, Correct: 129/150, Acc: 0.860000
Epoch 94, MSE: 0.037828, Correct: 129/150, Acc: 0.860000
Epoch 95, MSE: 0.037603, Correct: 129/150, Acc: 0.860000
Epoch 96, MSE: 0.037380, Correct: 129/150, Acc: 0.860000
Epoch 97, MSE: 0.037157, Correct: 129/150, Acc: 0.860000
Epoch 98, MSE: 0.036937, Correct: 129/150, Acc: 0.860000
Epoch 99, MSE: 0.036717, Correct: 130/150, Acc: 0.866667
Epoch 100, MSE: 0.036500, Correct: 130/150, Acc: 0.866667

MNIST Dataset

Configurations:

Epochs: 30 Hidden nodes: 30 Batchsize: 10 eta/learning rate: 3

Matlab Code:

```
>> load('mnistTrn.mat');
>> MyNetwork(trn, trnAns, [784, 30, 10], 30, 10, 3);
```

Matlab Output:

```
Epoch 1, MSE: 0.049364, Correct: 5376/50000, Acc: 0.107520
Epoch 2, MSE: 0.049113, Correct: 5690/50000, Acc: 0.113800
Epoch 3, MSE: 0.048822, Correct: 6070/50000, Acc: 0.121400
Epoch 4, MSE: 0.048459, Correct: 6590/50000, Acc: 0.131800
Epoch 5, MSE: 0.048026, Correct: 7322/50000, Acc: 0.146440
Epoch 6, MSE: 0.047567, Correct: 8052/50000, Acc: 0.161040
Epoch 7, MSE: 0.047125, Correct: 8728/50000, Acc: 0.174560
Epoch 8, MSE: 0.046700, Correct: 9389/50000, Acc: 0.187780
Epoch 9, MSE: 0.046286, Correct: 10009/50000, Acc: 0.200180
Epoch 10, MSE: 0.045890, Correct: 10531/50000, Acc: 0.210620
Epoch 11, MSE: 0.045525, Correct: 11015/50000, Acc: 0.220300
Epoch 12, MSE: 0.045195, Correct: 11478/50000, Acc: 0.229560
Epoch 13, MSE: 0.044898, Correct: 11833/50000, Acc: 0.236660
Epoch 14, MSE: 0.044627, Correct: 12157/50000, Acc: 0.243140
Epoch 15, MSE: 0.044380, Correct: 12454/50000, Acc: 0.249080
Epoch 16, MSE: 0.044162, Correct: 12804/50000, Acc: 0.256080
Epoch 17, MSE: 0.043974, Correct: 13172/50000, Acc: 0.263440
Epoch 18, MSE: 0.043820, Correct: 13599/50000, Acc: 0.271980
Epoch 19, MSE: 0.043695, Correct: 14066/50000, Acc: 0.281320
Epoch 20, MSE: 0.043598, Correct: 14453/50000, Acc: 0.289060
Epoch 21, MSE: 0.043523, Correct: 14775/50000, Acc: 0.295500
Epoch 22, MSE: 0.043465, Correct: 15061/50000, Acc: 0.301220
Epoch 23, MSE: 0.043421, Correct: 15303/50000, Acc: 0.306060
Epoch 24, MSE: 0.043387, Correct: 15512/50000, Acc: 0.310240
Epoch 25, MSE: 0.043362, Correct: 15674/50000, Acc: 0.313480
Epoch 26, MSE: 0.043342, Correct: 15845/50000, Acc: 0.316900
Epoch 27, MSE: 0.043326, Correct: 15992/50000, Acc: 0.319840
Epoch 28, MSE: 0.043313, Correct: 16097/50000, Acc: 0.321940
Epoch 29, MSE: 0.043300, Correct: 16180/50000, Acc: 0.323600
Epoch 30, MSE: 0.043289, Correct: 16293/50000, Acc: 0.325860
```

XOR Problem

Configurations:

Epochs: 10 Hidden nodes: 2 Batchsize: 4 eta/learning rate: 0.1

Matlab Code:

```
>> input = [1 1 0 0; 1 0 1 0];
>> target = [0 1 1 0];
>> MyNetwork(input, target, [2, 2, 1], 10, 4, 0.1);
```

Matlab Output:

```
Epoch 1, MSE: 0.154652, Correct: 2/4, Acc: 0.500000
```

Epoch 2, MSE: 0.154313, Correct: 2/4, Acc: 0.500000
Epoch 3, MSE: 0.153976, Correct: 2/4, Acc: 0.500000
Epoch 4, MSE: 0.153642, Correct: 2/4, Acc: 0.500000
Epoch 5, MSE: 0.153310, Correct: 2/4, Acc: 0.500000
Epoch 6, MSE: 0.152981, Correct: 2/4, Acc: 0.500000
Epoch 7, MSE: 0.152654, Correct: 2/4, Acc: 0.500000
Epoch 8, MSE: 0.152330, Correct: 2/4, Acc: 0.500000
Epoch 9, MSE: 0.152008, Correct: 2/4, Acc: 0.500000
Epoch 10, MSE: 0.151689, Correct: 2/4, Acc: 0.500000

Configurations:
Epochs: 10 Hidden nodes: 2 Batchsize: 1 eta/learning rate: 0.1
Matlab Code:
>> MyNetwork(input, target, [2, 2, 1], 10, 1, 0.1);
Matlab Output:
Epoch 1, MSE: 0.152239, Correct: 2/4, Acc: 0.500000
Epoch 2, MSE: 0.151766, Correct: 2/4, Acc: 0.500000
Epoch 3, MSE: 0.151296, Correct: 2/4, Acc: 0.500000
Epoch 4, MSE: 0.150831, Correct: 2/4, Acc: 0.500000
Epoch 5, MSE: 0.150369, Correct: 2/4, Acc: 0.500000
Epoch 6, MSE: 0.149912, Correct: 2/4, Acc: 0.500000
Epoch 7, MSE: 0.149458, Correct: 2/4, Acc: 0.500000
Epoch 8, MSE: 0.149009, Correct: 2/4, Acc: 0.500000
Epoch 9, MSE: 0.148565, Correct: 2/4, Acc: 0.500000
Epoch 10, MSE: 0.148124, Correct: 2/4, Acc: 0.500000
Configurations:
Epochs: 20 Hidden nodes: [3 2] Batchsize: 1 eta/learning rate: 0.1
Matlab Code:
>> MyNetwork(input, target, [2, 3, 2, 1], 20, 1, 0.1);
Matlab Output:
Epoch 1, MSE: 0.128475, Correct: 2/4, Acc: 0.500000
Epoch 2, MSE: 0.128231, Correct: 2/4, Acc: 0.500000
Epoch 3, MSE: 0.128000, Correct: 2/4, Acc: 0.500000
Epoch 4, MSE: 0.127779, Correct: 2/4, Acc: 0.500000
Epoch 5, MSE: 0.127568, Correct: 2/4, Acc: 0.500000
Epoch 6, MSE: 0.127368, Correct: 2/4, Acc: 0.500000
Epoch 7, MSE: 0.127177, Correct: 2/4, Acc: 0.500000
Epoch 8, MSE: 0.126995, Correct: 2/4, Acc: 0.500000
Epoch 9, MSE: 0.126822, Correct: 2/4, Acc: 0.500000
Epoch 10, MSE: 0.126657, Correct: 2/4, Acc: 0.500000
Epoch 11, MSE: 0.126501, Correct: 2/4, Acc: 0.500000
Epoch 12, MSE: 0.126352, Correct: 2/4, Acc: 0.500000
Epoch 13, MSE: 0.126210, Correct: 2/4, Acc: 0.500000
Epoch 14, MSE: 0.126075, Correct: 2/4, Acc: 0.500000

Epoch 15, MSE: 0.125947, Correct: 2/4, Acc: 0.500000
Epoch 16, MSE: 0.125825, Correct: 2/4, Acc: 0.500000
Epoch 17, MSE: 0.125709, Correct: 2/4, Acc: 0.500000
Epoch 18, MSE: 0.125599, Correct: 2/4, Acc: 0.500000
Epoch 19, MSE: 0.125494, Correct: 2/4, Acc: 0.500000
Epoch 20, MSE: 0.125395, Correct: 2/4, Acc: 0.500000

Overall, we see that with the neural network implementation I have currently, the algorithm can still be enhanced to improve the accuracy.