Homework 4: Counterprop

Andy Reagan

A basic version of the counterpropogation algorithm is coded up, and put through some simple tests. I find that it performs best when training with Kohonen and Grossberg layers simulanteously, when the learned parameters are not scaled, and when run from random initial weights (as opposed to nearest neighbor).

I. INTRODUCTION

The original version of the counterpropogation algorithm was originally proposed by Neilson in 1988 [1] with a later paper describing the applications [2]. Here I code up a version using the psuedocode from Rizzo and Doughtery's application to characterizing aquifer properties [3].

The main idea of counterpropogation is a combination of the classification power of Kohonen's map, with the interpolation ability of a Grossberg net. Claimed advantages of counterpropogation are higher speed than backpropogation, and the ability to better recognize patterns.

As I have found to be the case with coding the nueral networks, there are many parameters and choices that are left up to the programmer. In the next section we explore a few, namely: order of presentation of training data, effect of training the layers together or separate, on scaling the learning coefficients, and on initializing the weights.

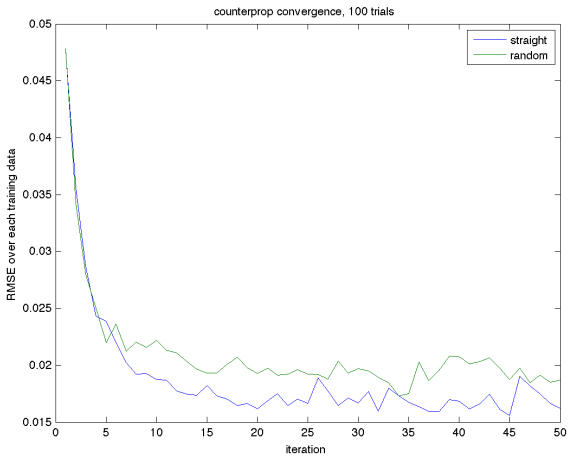

FIG. 1: Direct comparison over 100 trials of the convergence when presenting training data in a random or repeated order.

II. METHODS

I code the network in MATLAB, again following the psuedocode from Rizzo and Doughtery [3]. Specific choices of parameters are investigated further.

I set the number of Kohonen nuerons to be 3 greater than the size of the training data when not using the network as a nearest-neighbor classifier. This is such that the Kohonen vectors can be useful in a nearest-neighbor classification fashion, but without prior assumptions about the distributions of the given observations. Another interesting thing to try would be to explicitly remove the Kohonen nodes whose input weights go to zero, and therefore use the Kohonen layer to reduce the data into clusters. I don't expect that removing these nodes would have any effect, but it would be good to check.

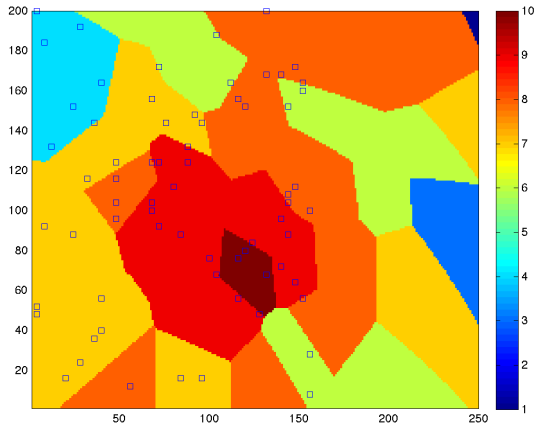

FIG. 2: Classification of full data (interpolation phase), with points at where the training data were.

III. RESULTS

First, the main challenge of getting the network to run, and converge, was successful. Both operating from a random initial condition on the weights on and on a preset nearest neighbor condition, we are able to make spatially reasonable predictions for the training data.

As an interpolater, the Grossberg layer classifies the "fuzzy" output of the Kohonen layer as a winner-takes-all, and this seems like a reasonable and simple way to classify this output.

I find that despite my expectations, the best convergence

Typeset by REVTEX

is achieved without scaling parameters. The best performance is also achieved while training together, and agian best performance is achieved on initializing the weights randomly, and letting the Kohonen layer do it's thing. Intiutively, as a spatial classifier, I really like counterpropogation.

Each of the following figures is the result of a test of each of these over 100 trials. I apologize for the tiny font size, MATLAB default.

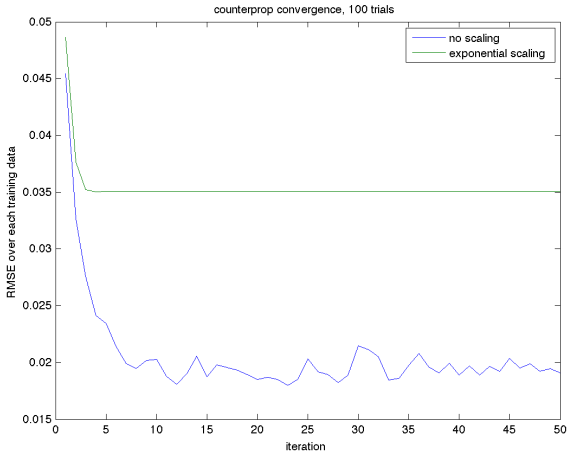

FIG. 3: Convergence with different scalings of learning parameters.

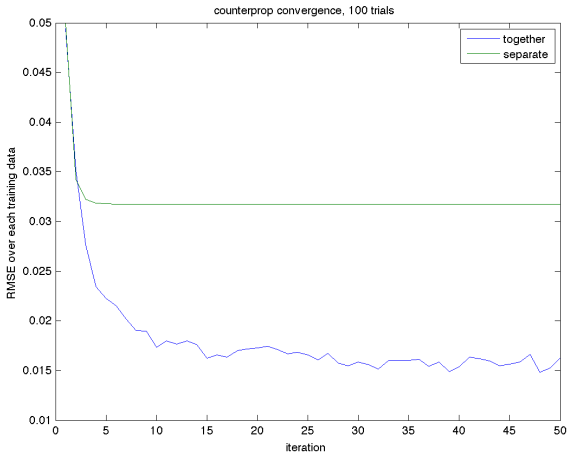

FIG. 4: Covergence training together and separate.

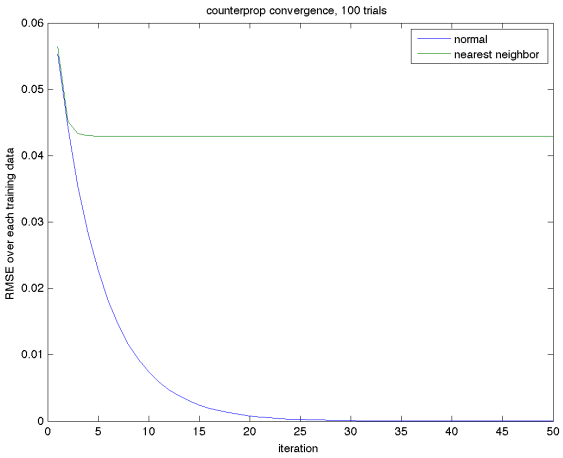

FIG. 5: Covergence with different Kohonen layer weight initializations.

[1] Hecht-Nielsen, R. (1987). Counterpropagation networks. *Applied optics 26*(23), 4979–4983.

[2] Hecht-Nielsen, R. (1988). Applications of counterpropagation networks. *Neural networks 1*(2), 131–139.

[3] Rizzo, D. M. and D. E. Dougherty (1994). Characterization of aquifer properties using artificial neural networks: neural kriging. *Water Resources Research 30*(2), 483–497.

Full code

```
% clear all
% close all

% load the data
training = csvread('CounterProp_Data.csv');

num_output_categories = max(training(:,end));
output_data = zeros(length(training(:,1)),num_output_categories);
for i=1:length(training(:,1))
    output_data(i,end+1-training(i,end)) = 1;
end

% make the interpolation data set
% looks like the x and y in the training data
% go close to 200 and 250, respectively
% so interpolation at every point up to those
[X,Y] = meshgrid(1:200,1:250);
interpolation = [X(:),Y(:)];

% normalize the data to the unit sphere
[training_norm,interpolation_norm] = nomalize_input_andy(training(:,1:end-1),interpolation);

% tell me what happened
fprintf('size of training is: \n');
disp(size(training));
fprintf('size of training_norm is: \n');
disp(size(training_norm));
fprintf('first row of training is: \n');
disp(training(1,:));
fprintf('first row of training_norm is: \n');
disp(training_norm(1,:));
fprintf('length of first row of training is (should be 1): \n');
disp(sum(training_norm(1,:).^2));
fprintf('length of first row of interpolation is (should be 1): \n');
disp(sum(interpolation_norm(1,:).^2));

numiter = 50;
% my algorithm breaks down for too many iterations....
% if I put 50, it sometimes works and sometimes doesn't
% fixed! had to use find(...,1)

% train the weight matrices
[~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,0,@scaling_none,true,false);
% again, tell me what happened
% disp(size(V));
% disp(size(W));

figure(111)
plot(1:numiter,mean(errors_all,1))
title('convergence of counterprop')
xlabel('iteration')
ylabel('avg RMSE over training data')
saveas(111,'111.png')

figure(112)
plot(1:numiter,errors_all)
title('convergence of counterprop')
xlabel('iteration')
ylabel('RMSE over each training data')
saveas(112,'112.png')

[W,V,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,1,@scaling_none,true,false);

figure(113)
plot(1:numiter,mean(errors_all,1))
title('convergence of counterprop, random training order')
xlabel('iteration')
ylabel('avg RMSE over training data')
saveas(113,'113.png')

figure(114)
plot(1:numiter,errors_all)
title('convergence of counterprop, random training order')
xlabel('iteration')
ylabel('RMSE over each training data')
saveas(114,'114.png')

% looking at a few of these...looks like it flattens out
% with the random order
```

```
% and the avg RMSE is about the same

% let's do a little bit better, and average over many runs for the
% RMSE plot
numtrials = 10;
errors_randomVstraight = zeros(2,numiter);
for i=1:numtrials
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,0,@scaling_none,true,false);
    errors_randomVstraight(1,:) = mean(errors_all,1);
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,0,@scaling_none,true,false);
    errors_randomVstraight(2,:) = mean(errors_all,1);
end

errors_randomVstraight = errors_randomVstraight./numtrials;

figure(115)
plot(1:numiter,errors_randomVstraight)
title('counterprop convergence, 100 trials')
xlabel('iteration')
ylabel('RMSE over each training data')
legend('straight','random')
saveas(115,'115.png')

numiter = 10;

% test it now
% train one with random order
randorder = true;
[kohonen_weights,grossberg_weights,~] = train_counterprop_andy(training_norm,output_data,numiter,randorder,
    @scaling_none,true,false);

test_counterprop_andy(kohonen_weights,grossberg_weights,interpolation_norm,training_norm,training,'116-standard.
    png',false);
test_counterprop_andy(kohonen_weights,grossberg_weights,interpolation_norm,training_norm,training,'116-wpoints.png
    ',true);

% now lets try to look at convergence with exponential scaling
numiter = 50;
errors_exp = zeros(2,numiter);
for i=1:numtrials
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,false,@scaling_none,true,false);
    errors_exp(1,:) = mean(errors_all,1);
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,false,@scaling_exponential,true,
        false);
    errors_exp(2,:) = mean(errors_all,1);
end

errors_exp = errors_exp./numtrials;

figure(117)
plot(1:numiter,errors_exp)
title('counterprop convergence, 100 trials')
xlabel('iteration')
ylabel('RMSE over each training data')
legend('no scaling','exponential scaling')
saveas(117,'117.png')

% now lets try to look at convergence separately vs together
errors_exp = zeros(2,numiter);
for i=1:numtrials
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,false,@scaling_none,true,false);
    errors_exp(1,:) = mean(errors_all,1);
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,false,@scaling_exponential,false,
        false);
    errors_exp(2,:) = mean(errors_all,1);
end

errors_exp = errors_exp./numtrials;

figure(118)
plot(1:numiter,errors_exp)
title('counterprop convergence, 100 trials')
xlabel('iteration')
ylabel('RMSE over each training data')
legend('together','separate')
saveas(118,'118.png')

% now lets try to look at convergence w nearest neighbor
numiter = 50;
errors_exp = zeros(2,numiter);
for i=1:numtrials
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,false,@scaling_none,false,true);
    errors_exp(1,:) = mean(errors_all,1);
```

```
    [~,~,errors_all] = train_counterprop_andy(training_norm,output_data,numiter,false,@scaling_exponential,false,
        true);
    errors_exp(2,:) = mean(errors_all,1);
end

errors_exp = errors_exp./numtrials;

figure(119)
plot(1:numiter,errors_exp)
title('counterprop convergence, 100 trials')
xlabel('iteration')
ylabel('RMSE over each training data')
legend('normal','nearest neighbor')
saveas(119,'119.png')

% test randomorder on the 2D space
numiter = 50;
% i'm not sure averaging over weights makes any real sense
% averaging over convergence does, but not weights
numtrials = 1;
% for i=1:numtrials
randorder = true;
[kohonen_weightsr,grossberg_weightsr,~] = train_counterprop_andy(training_norm,output_data,numiter,randorder,
    @scaling_none,true,false);
randorder = false;
[kohonen_weights,grossberg_weights,~] = train_counterprop_andy(training_norm,output_data,numiter,randorder,
    @scaling_none,true,false);
% end
test_counterprop_andy(kohonen_weightsr,grossberg_weightsr,interpolation_norm,training_norm,training,'interpolation
    -randorder.png',false);
test_counterprop_andy(kohonen_weights,grossberg_weights,interpolation_norm,training_norm,training,'interpolation-
    straigtorder.png',true);

% test nearest neighbor in 2d
numiter = 50;
randorder = false;
[kohonen_weightsr,grossberg_weightsr,~] = train_counterprop_andy(training_norm,output_data,numiter,randorder,
    @scaling_none,false,false);
[kohonen_weights,grossberg_weights,~] = train_counterprop_andy(training_norm,output_data,numiter,randorder,
    @scaling_none,false,true);
test_counterprop_andy(kohonen_weightsr,grossberg_weightsr,interpolation_norm,training_norm,training,'interpolation
    -random.png',false);
test_counterprop_andy(kohonen_weights,grossberg_weights,interpolation_norm,training_norm,training,'interpolation-
    nearestn.png',true);
```

```
function [B,C] = nomalize_input_andy(training_patterns,interpolation_patterns)
% normalize the input using the unit sphere method
%
% none of this is vectorized....we'll wait and see if it needs to be
% ......did it anyway, the interpolation data is huge right now
%
% INPUTS
%
% training_patterns(num train patterns, size input)
% interpolation_patterns(num interp patterns, size input)
%
% OUTPUT
%
% B: normalized input

num_training_patterns = length(training_patterns(:,1));
input_size = length(training_patterns(1,:));

% B = zeros(num_training_patterns,input_size+1);
% for i=1:num_training_patterns
%     B(i,1:input_size) = training_patterns(i,:);
%     % this is the "l" from the notes
%     B(i,input_size+1) = sqrt(sum(training_patterns(i,:).^2));
% end
B = ones(num_training_patterns,input_size+1);
B(:,1:end-1) = training_patterns;
B(:,end) = sqrt(sum(training_patterns.^2,2));

num_interpolation_patterns = length(interpolation_patterns(:,1));
C = ones(num_interpolation_patterns,input_size+1);
C(:,1:end-1) = interpolation_patterns;

% l_interpolation = zeros(num_interpolation_patterns,1);
% for i=1:num_interpolation_patterns
%     % this is the "l" from the notes
%     l_interpolation(i) = sqrt(sum(interpolation_patterns(i,:).^2));
% end
```

```
C(:,end) = sqrt(sum(interpolation_patterns.^2,2));

max_l = max([max(B(:,end)),max(C(:,end))]);
N = 1.01 * max_l;

B(:,end) = sqrt(N^2-B(:,end).^2);
C(:,end) = sqrt(N^2-C(:,end).^2);

B = B/N;
C = C/N;

end
```

```
function [W,V,rmse_all] = train_counterprop_andy(A,B,numiter,randorder,scaling_fun,traintogether,nearest_neighbor)
% train the two counterprop matrices
%
% INPUTS
%
% A(num train patterns,size input)
% B(num train patterns,size output)
%
% OUTPUT
%
% V: Kohonen layer weights
% W: Grossberg layer weights

% how long to train
% numiter = 50;
% taking this from input

% hardcode learning coeff
alpha = 0.7; % kohonen
beta = 0.2; % grossberg


num_training_patterns = length(A(:,1));

% rmse_all = ones(num_training_patterns+(1-traintogether)* ...
%                 num_training_patterns,numiter);
rmse_all = ones(num_training_patterns,numiter);
% rmse_avg = ones(1,numiter);

input_size = length(A(1,:));
output_size = length(B(1,:));

% kohonen later
if nearest_neighbor
    hidden_layer_size = num_training_patterns;
    W = A';
else
    hidden_layer_size = num_training_patterns+3;
    W = rand(input_size,hidden_layer_size);
end
% disp(size(W));
% grossberg layer
V = rand(hidden_layer_size,output_size);

if ~nearest_neighbor
    fprintf('training␣kohonen␣layer\n');
    for i=1:numiter
        % fprintf('on training iteration no:\n');
        % disp(i);
        if randorder
            order = 1:num_training_patterns;
        else
            order = randperm(num_training_patterns);
        end
        for j=order
            % apply kohonen weights to compute middle layer
            % in your paper, figure 4 shows that this should compute
            % the minumim distance between input and kohonen layer
            % weights...
            % this just applies the weights forward:
            I = A(j,:)*W;
            % and this is more akin the distance
            % (1x3)*(3x78) = (1x78)
            % I = 1-A(j,:)*W;
            % NOTE: these are the "z_j" in the psuedocode

            % find the index of min
            % if we didn't take 1-... in the above, could take the max:
            winning_node = find(I==max(I),1);
```

```
            % this variable is akin to "c" in the code
            % winning_node = find(I==min(I),1);

            % update this just to look at in debugging
            % I = I>=max(I);

            % disp(I);
            % disp(winning_node);

            % update the winning node's links in kohonen layer
            % this should move the weights toward the input

            % fprintf('input');
            % disp(A(j,:)');
            % fprintf('winning row');
            % disp(W(:,winning_node));
            W(:,winning_node) = W(:,winning_node) + scaling_fun(alpha,i)*(A(j,:)'-W(:,winning_node));
            % fprintf('winning row after');
            % disp(W(:,winning_node));

            % update links in grossberg layer
            % note that y' is just V(winning_node,:)
            % since the a_j is set to 1
            if traintogether
                % save the error in the output
                output_error = B(j,:)-V(winning_node,:);

                rmse_all(j,i) = rmse(B(j,:),V(winning_node,:));
                V(winning_node,:) = V(winning_node,:) + scaling_fun(beta,i)*(output_error);
            end
        end
        % rmse_avg(1,i) = mean(rmse_all(:,i));
    end
end

% train the grossberg layer
if ~traintogether || nearest_neighbor
    fprintf('training just grossberg layer now\n');
    for i=1:numiter
        if randorder
            order = 1:num_training_patterns;
        else
            order = randperm(num_training_patterns);
        end
        for j=order
            I = 1-A(j,:)*W;
            winning_node = find(I==min(I),1);
            output_error = B(j,:)-V(winning_node,:);
            rmse_all(j,i) = rmse(B(j,:),V(winning_node,:));
            V(winning_node,:) = V(winning_node,:) + scaling_fun(beta,i)*(output_error);
        end
    end
end

end
```

```
function [fullgrid] = test_counterprop_andy(kohonen,grossberg,interpolation,training,training_raw,f,plottraining)
% something something something

% define the full grid to test
% xmax = max(interpolation(:,1));
% ymax = max(interpolation(:,2));
% oops, those are normalized!!
xmax = 200;
ymax = 250;
fullgrid = ones(xmax,ymax);

winner_take_all = @(x) x>=(max(x));

for x=1:xmax
    for y=1:ymax
        % this is non vectorized
        % % indexing!
        % A = interpolation(x*ymax-ymax+y,:);
        % % apply kohonen layer
        % I = 1-A*kohonen;
        % % threshold
        % I = winner_take_all(I);
        % % apply grossberg layer
        % Y = I*grossberg;
        % % threshold
        % Y = winner_take_all(Y);
```

```
        % % take the number from the output!
        % fullgrid(x,y) = find(Y==max(Y));

        % this is vectorized
        % but redundant
        % fullgrid(x,y) = find(winner_take_all(1-interpolation(x*ymax-ymax+y,:)*kohonen)*grossberg==max(
            winner_take_all(1-interpolation(x*ymax-ymax+y,:)*kohonen)*grossberg));
        % better
        B = winner_take_all(interpolation(x*ymax-ymax+y,:)*kohonen)*grossberg;
        fullgrid(x,y) = find(B==max(B));
    end
end

% i am very aware that this is missing a row when using flat
% shading
% but I don't feel like fighting with matlab
figure(116);
pcolor(fullgrid);
shading flat;
colorbar;
if plottraining
    hold on;
    for i=1:length(training_raw)
        plot(training_raw(i,1),training_raw(i,2),'s');
    end
end
saveas(116,f);


end


function v = scaling_exponential(param,iter)
v = param^iter;


function v = scaling_none(param,~)
v = param;
```

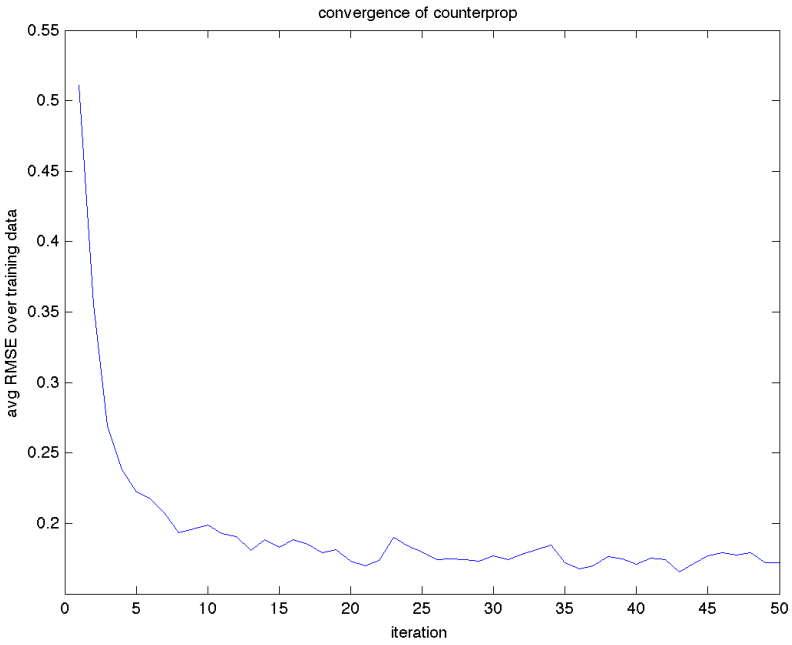

FIG. 6: Basic convergence test.

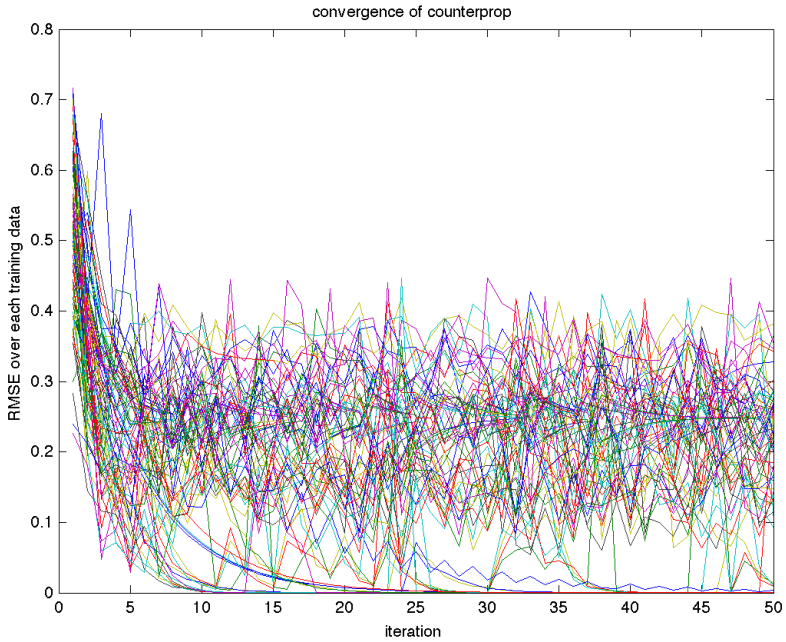

FIG. 7: Convergence on each of the individual training data.

Extra figures

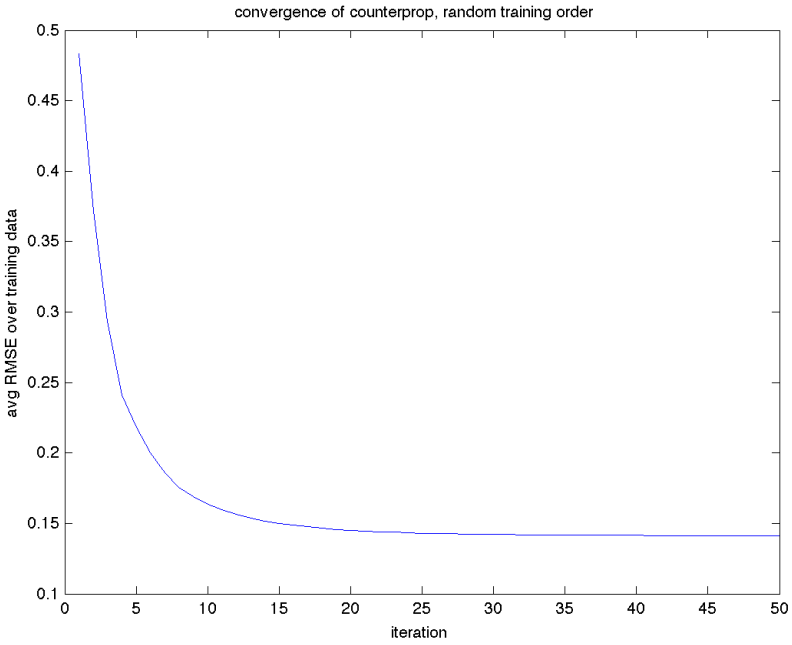

FIG. 8: Randomized order of presentation of the training data.

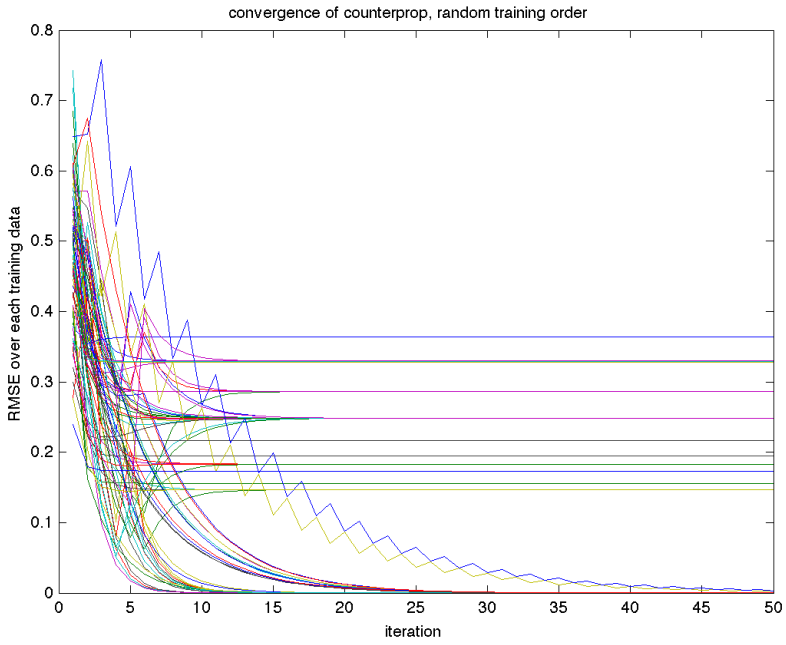

FIG. 9: Randomized order of presentation of the training data, and convergence on each of the training data.

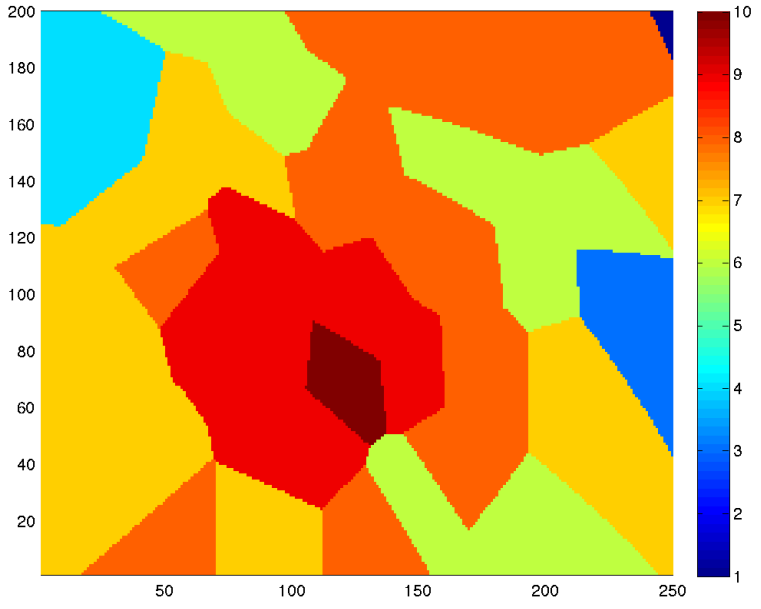

FIG. 10: Classification of full data (interpolation phase).