Homework 5: SOM

Andy Reagan

I code and discuss an implementation of Kohonen's SOM as described in his 1990 review.

I. INTRODUCTION

Originally proposed by Kohonen, the self organizing map describes a general class of unsupervised artificial nueral networks [1]. They can be run unsupervised, but in use as a classifier it is best to supervise a fine tuning process, and they become a semi-supervised process.

II. METHODS

I closely follow the paper of Kohonen in coding up the SOM. It is definitely tricky, but leading the paper discussion forced me to understand it a bit more.

Since I'm hoping to use this as a classifier in my project, I'm paid careful attention to this use case. In particular, there are two ways to lay out the network. The first, in displaying how the animals are classified, looks at the nodes of the network laid in space, by their topology. The second lays out the nodes by their values, giving a better sense of the distribution which they are now approximating.

The big hang-up for my code is implementing the network topology, and changing the size of the neighborhood used in training. I took the flat network, and built an adjacency matrix for it. Then, based how far in the training we are, I travel this adjacency matrix for across a maximum path length. My implementations of both of these concepts are definitely lacking.

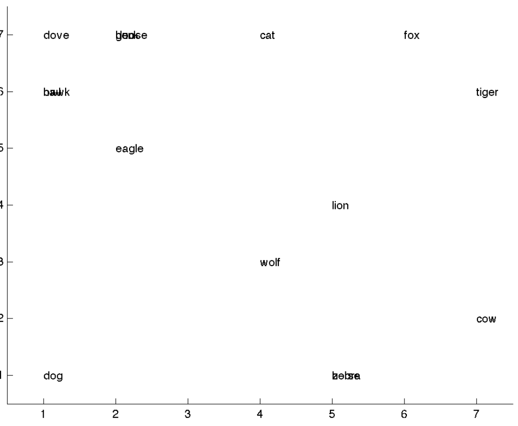

FIG. 1: Labels for different animals plotted over their closest Kohonen node. The nodes are laid out on a square grid, which is like the topology of the network, which uses a Von Neumann neighborhood. Some animals have the same attributes, and got plotted in the same place!

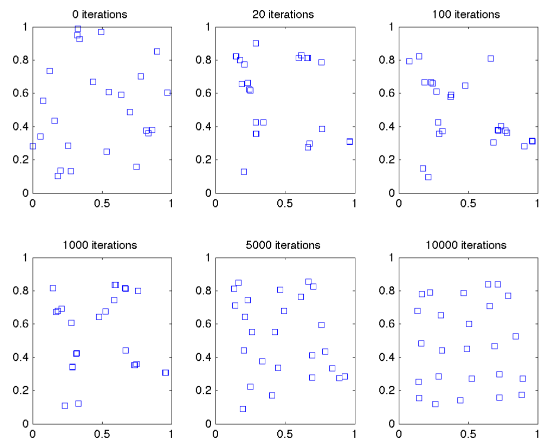

FIG. 2: The best I could get a small SOM to cover the uniform distribution in 2D.

Thinking about the network like a CA, it might be easier to use a matrix as the nodes and avoid the full adjacency matrix altogether. This would definitely be faster. But this approach in limited in the type of topology that is possible.

III. RESULTS

I had no problems, over a variety of parameters, to do a reasonably good job separating out the types of animals. Attempting to get the map to spread out over the uniform 2D distribution was more difficult. My implementation is slow, so I was only able to run reasonably for 100 samples and a network of size 25. It was sensitive to both the neighborhood decay and the decay of the learning parameter. A decay that was too rapid in the learning parameter didn't give it enough time to spread out, and a deacy that was too slow let it spread out, then cluster again. It was also important to keep the path length long initially, for it to spread out. And if that doesn't decay quick enough, my code takes forever to run.

In the end, I think that the SOM is a lot of fun, and is definitely worthwhile for different applications. But, you need to be careful about how to apply it, and how

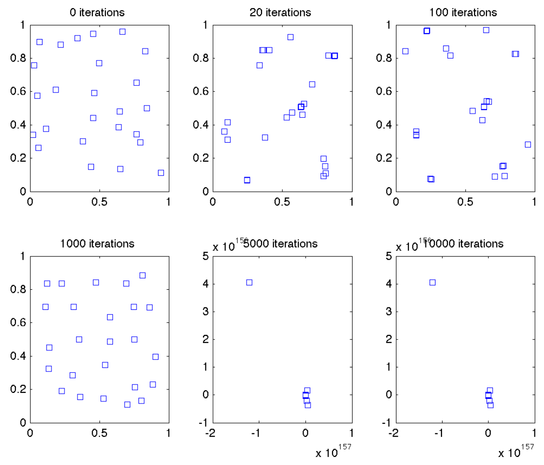

FIG. 3: An example where the learning parameter decays too slowly. What looks like convergence is lost! Notice that the bounds on the final two plots grow to 10^5.

to interperet the results. In particular, although it is "unsupervised", it's not a black box. So, you still have to know what's going on, and be careful, like with the fully supervised learning that we've seen.

What follows is essentially restatement of what Kohonen says in his paper, and an inclusion of the equation for the Bayes classification, trying to make sense of how this works. As a bayesian classifier, we write the conditional probability of the vector x belonging to class C_k as

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)} \tag{1}$$

and, as Kohonen writes, the decision boundaries are where the function

$$|p(C_i)p(x|C_i) - p(C_j)p(x|C_j)| \tag{2}$$

eqauls 0. So, by using the nearest neighbor choice for classification, this is akin to using the bayesian classifier

$$k = \max_k p(C_k) \prod_i p(x_i|C_k) \tag{3}$$

where we're choosing the k such that our network nodes have the greatest density in $p(x_i|C_k)$.

[1] Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE 78*(9), 1464–1480.

Full code

```
% clear all
% close all

% load the data, into the variable X
load AnimalData.mat
A = X';

% % 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(0:.01:1,0:01:1);
% interpolation = [X(:),Y(:)];

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

numiter = 150;
randorder = 1;

% initialize the kohonen weights
% these are the m_i
num_nodes = 49;
B = rand(length(A(:,1)),num_nodes);

node_network_size = [7,7]; % down x across

% create the adjacency matrix
C = zeros(num_nodes,num_nodes);
for i=1:node_network_size(1) % down
    for j=1:node_network_size(2) % across
        indices = unique([sub2ind(node_network_size,i,min([node_network_size(2),j+1])),sub2ind(node_network_size,i
            ,max([1,j-1])),sub2ind(node_network_size,max([1,i-1]),j),sub2ind(node_network_size,min([i+1,
            node_network_size(1)]),j)]);
        % disp(indices);
        C(sub2ind(node_network_size,i,j),indices) = 1;
        C(indices,sub2ind(node_network_size,i,j)) = 1;
        C(sub2ind(node_network_size,i,j),sub2ind(node_network_size,i,j)) =1;
    end
end

% train the weight matrices
[B,errors_all] = train_SOM(A,B,C,numiter,randorder,@scaling_inverse,@moore_decaying,0);

figure(111)
titles = {'dove','hen','duck','goose','owl','hawk','eagle','fox','dog','wolf','cat','tiger','lion','horse','zebra'
    ,'cow'};
for i=1:length(A(1,:))
    min_dist = sqrt(sum((A(:,i)-B(:,1)).^2));
    winning_node = 1;
    for k=2:num_nodes
        dist = sqrt(sum((A(:,i)-B(:,k)).^2));
        if dist<min_dist
            min_dist = dist;
            winning_node = k;
        end
    end
    fprintf('winning␣node␣is␣%f\n',winning_node);
    [nodei,nodej] = ind2sub(node_network_size,winning_node);
    text(nodei,nodej,sprintf('%s',titles{i}))
    hold on;
end
xlim([0.5 node_network_size(1)+.5])
ylim([0.5 node_network_size(2)+.5])

% plot(1:numiter,mean(errors_all,1))
% title('convergence of SOM')
% xlabel('iteration')
% ylabel('avg distance to training data')
saveas(111,'figures/animals-layout-corrected.png')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% here let's do the uniform distribution in 2D
% reproducing figure 3 of kohonen
numsamples = 100;
A = rand(2,numsamples);

numiter = 10;
randorder = 1;
```

```
% initialize the kohonen weights
% these are the m_i
num_nodes = 25;
B = rand(length(A(:,1)),num_nodes);

node_network_size = [5,5]; % down x across

% create the adjacency matrix
C = zeros(num_nodes,num_nodes);
for i=1:node_network_size(1) % down
    for j=1:node_network_size(2) % across
        indices = unique([sub2ind(node_network_size,i,min([node_network_size(2),j+1])),sub2ind(node_network_size,i
            ,max([1,j-1])),sub2ind(node_network_size,max([1,i-1]),j),sub2ind(node_network_size,min([i+1,
            node_network_size(1)]),j)]);
        % disp(indices);
        C(sub2ind(node_network_size,i,j),indices) = 1;
        C(indices,sub2ind(node_network_size,i,j)) = 1;
        C(sub2ind(node_network_size,i,j),sub2ind(node_network_size,i,j)) =1;
    end
end

iterations = [0,20,100,1000,5000,10000];
% iterations = [0,10,20,30,40,50];

figure(112);

i=1;
subplot(2,3,i);
plot(B(1,:),B(2,:),'s');
title(sprintf('%.0f␣iterations',iterations(i)))

% train the weight matrices
for i=2:length(iterations)
    [B,errors_all] = train_SOM(A,B,C,iterations(i)-iterations(i-1),randorder,@scaling_inverse,@moore_decaying,
        iterations(i-1));
    subplot(2,3,i);
    plot(B(1,:),B(2,:),'s');
    title(sprintf('%.0f␣iterations',iterations(i)))
end

saveas(112,'figures/SOM_uniform_dist_covering_long_corrected.png')
```

```
function [B,rmse_all] = train_SOM(A,B,C,numiter,randorder,scaling_fun,nbd_fun,iterstart)
% train the SOM neural net
%
% INPUTS
%
% A(size input,num training patterns)
% the training patterns
%
% B(size input,num nodes in kohonen)
% the m_i states of the nodes
%
% C(num nodes,num nodes)
% adjacency matrix for the nodes
%
% OUTPUT
%
% B: kohenen matrix values

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

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

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

fprintf('training\n');
for i=1+iterstart:numiter+iterstart
    fprintf('on␣training␣iteration␣no␣%f\n',i);
    if randorder
        order = 1:num_training_patterns;
    else
        order = randperm(num_training_patterns);
    end
    scaling_coeff = scaling_fun(i);
    fprintf('scaling␣coeff␣is␣%f\n',scaling_coeff);
    for j=order
        % find the index of the winner
        min_dist = sqrt(sum((A(:,j)-B(:,1)).^2));
        winning_node = 1;
        for k=2:num_nodes
            dist = sqrt(sum((A(:,j)-B(:,k)).^2));
            if dist<min_dist
                min_dist = dist;
                winning_node = k;
            end
        end
        % fprintf('winning node is %f\n',winning_node);
        rmse_all(j,i-iterstart) = min_dist;
        [nbd,nbd_coeffs] = nbd_fun(i,winning_node,C);
        % fprintf('tuning the nbd of size %f\n',length(nbd))
        for k=1:length(nbd)
            B(:,nbd(k)) = B(:,nbd(k)) - scaling_coeff*nbd_coeffs(k)*(B(:,nbd(k))-A(:,j));
        end
    end
    % rmse_avg(1,i) = mean(rmse_all(:,i));
end

end
```

```
function a = scaling_inverse(iter)
% scale the learning paramter
% per kohonen p.1469 point 2
a = 0.9*(1-iter/10000);
end
```

```
function [indices,coeff] = moore_decaying(iter,i,C)
% return the indices and the coefficients of the nbd of node i
%
% INPUT
%
% iter: iteration of the training
% i: index of the node
% C: adjacency matrix
%
% OUTPUT
% indices: indices of the neighbors
% coeffs: update coeffs of the neighbors

% convert the iteration to a path length
% ilen = 0.9*(1-iter/1000);
plen = max(1,floor(floor(length(C(1,:))/4)-iter*floor(length(C(1,:))/4)/10));

% display(plen);

indices = find(C(:,i)==1);
for j=2:plen
    l = length(indices);
    for k=1:l
        if ~isempty(find(C(:,indices(k))==1, 1))
            % disp(find(C(:,indices(k))==1));
            % indices = [indices;find(C(:,indices(k))==1)];
            indices = unique([indices;find(C(:,indices(k))==1)]);
        end
    end
end

% indices = unique(indices);

% don't scale them...
coeff = ones(size(indices));

end
```