Introduction to Intelligent Systems
Lab Session 4

Group 30
Sergio Calogero Catalano (s3540294) & Emmanouil Gionanidis (s3542068)

October 9, 2017

Unsupervised Learning - Vector Quantization

The winner-takes-all unsupervised competitive learning algorithm has been implemented and tested on the 3 data sets: the code written for this purpose is in **Appendix A** . When initializing prototypes, we decided to use random vectors from the example data, instead of random general vectors, to prevent some prototypes from being so far from the example data, that they would never win and, thus, never move. The results of the algorithm considerably depend on the number of prototypes K, on the number of epochs t_max, on the learning rate η, on the distance measure used and on the example data distribution. Thus we are going to discuss each of the previous key elements, especially the learning rate.

Number of prototypes

Vector quantization, in unsupervised learning, is mainly used to represent a large amount of data by some (possibly few) prototype vectors. The number of prototypes determines how precise such a representation will be: with few prototypes, many data examples will be approximated by one prototype; while with more prototypes there will be less data vectors approximated by the same prototype, making the representation more precise. However this does not mean that more prototypes imply a better representation, otherwise we could just use as many prototypes as the data vectors and we will have a perfect representation of the given data, with a quantization error of 0. The quality of a representation depends instead on the kind of data and on the goal of our unsupervised learning algorithm: therefore, sometimes, a relatively small number of prototypes describes the example data much better than a large amount of prototypes. This can be seen in the following plots, for which we used the data from w6_1x.mat, a learning rate of 0.01, a t_max of 150, a squared euclidean distance measure and different numbers of prototypes. When there are only 2 prototypes, there are points really far from their prototype, incrementing thus the quantization error; but as we increase the number of prototypes, the points become nearer to their prototypes and are thus represented more precisely by them (but not better, as previously discussed): intuitively a good number of prototypes for the plotted data is 2, but, depending on our goal, also other values can be chosen, such as 8, and the preciseness of the representation is not important in that choice. The figures **Figure 1** and **Figure 2** show the clusters obtained with 2 and 8 prototypes respectively: each cluster has a different color and the prototypes have bigger markers than the data points.

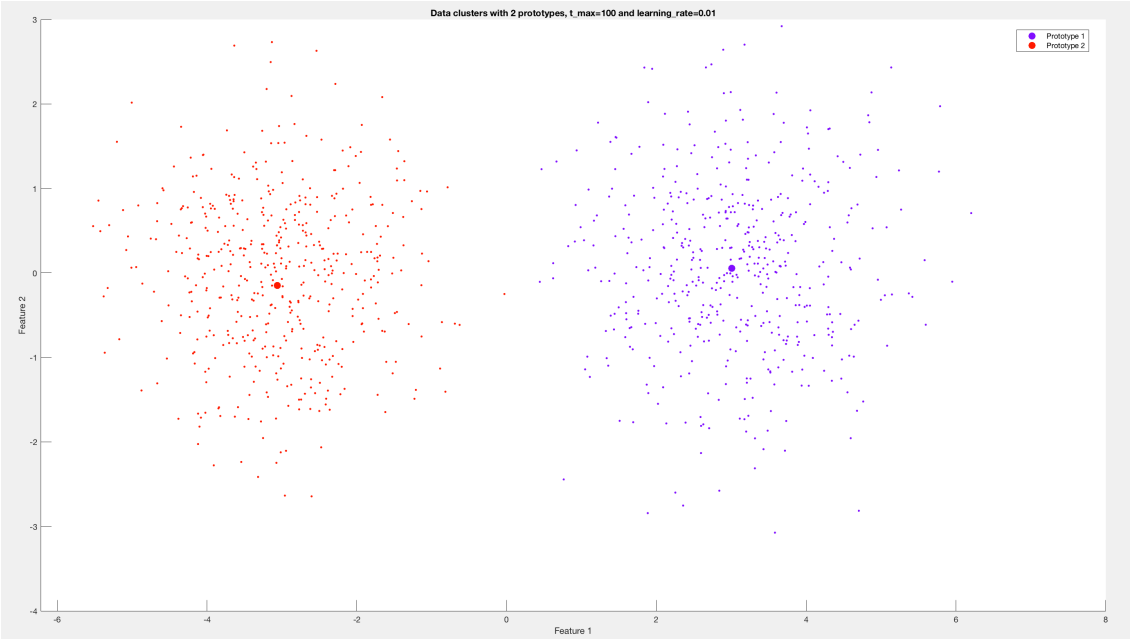

Figure 1: *shows the clusters obtained with 2 prototypes*

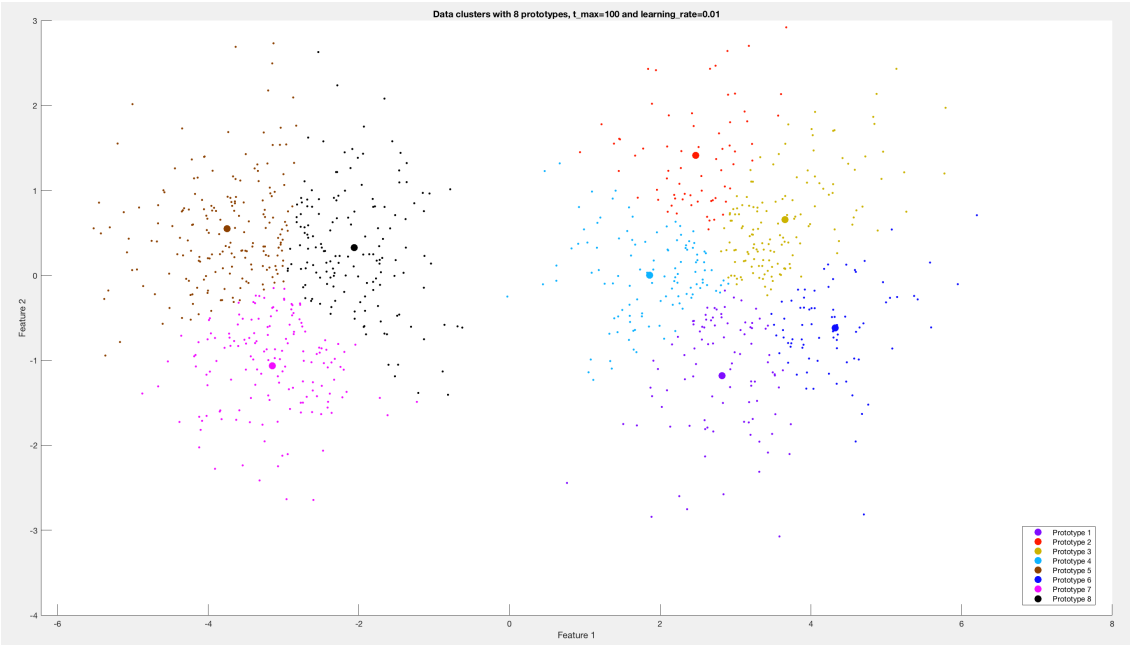

Figure 2: *shows the clusters obtained with 8 prototypes*

Considering what said until now, we have to find a way to choose an appropriate number of prototypes. This can be done by plotting the number of prototypes vs the quantization error and by choosing a value which represents the "elbow" of the obtained plot. **Figure 3** shows such a plot (with the same data, η, t_max and distance measure as the previous plots) and, using the "elbow rule", a good choice for the amount of prototypes is 2. But the quantization error decreases fast even after 2, so it would be good to consider also a slightly greater value, such as 4 or 6.

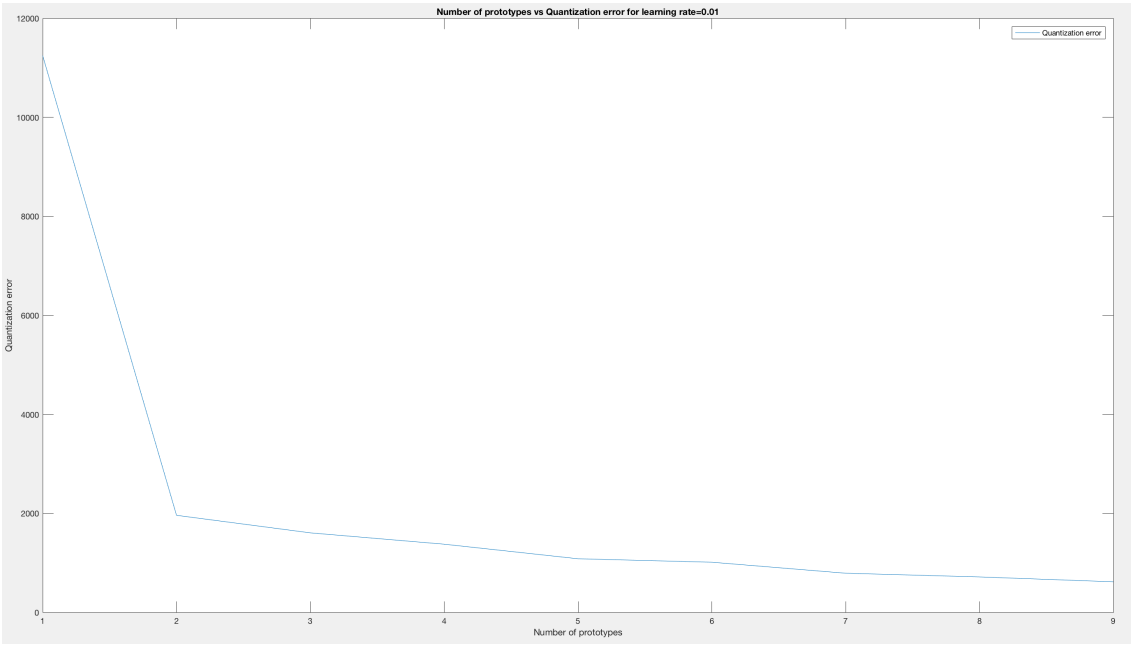

Figure 3: *shows the quantization error as a function of the number of prototypes*

Number of epochs

The number of epochs, generally, determines how near we get to a local minimum. However, with a sufficiently low learning rate, the quantization error stays approximately the same after a certain number of epochs. Therefore it is only necessary to choose that value as t_max. To be sure that we reached a good result, in the previous plots (and in most of the other plots as well) we used 100 epochs (which is good enough for values of η down to 10^{-6}). These observations can be verified with the plots provided in the learning rate section.

Learning rate

The learning rate is a fundamental part of the vector quantization algorithm, as it determines how much the prototypes move towards the example points at each iteration. With a low learning rate, more epochs are needed to reach a local minimum, but it is also more difficult to leave that minimum; with a large learning rate, it is very easy to leave a local minimum, with the possibility to reach a prototype configuration with a larger quantization error. Thus a too large value of η is not good, as the algorithm might never reach a local minimum(or more correctly, might not be in one when we terminate the algorithm), while a too small value of η requires more epochs to reach a local minimum and it is less likely to leave it. It is also possible to notice that, for high values of η, the plot does never stabilize and keeps floating around a value (this effect is probably enhanced by the use of the squared euclidean distance metric instead of the normal one): that explains why the standard deviation of the quantization error decreases when we decrease the learning rate.
Hence a good value for the learning rate would be one which does not produce any of the above effects: we want to reach a good local minimum (possibly the global one) in an acceptable amount of epochs and, if it is not a good minimum, we also want to leave it(if it is not a good local minimum, even a low value of η allows to leave it). We tried many combinations of parameters, and we show the plots which led us to the choice of a learning rate: **Figure 4** , **Figure 5** , **Figure 6** and **Figure 7** show plots of t(the epoch number) vs the quantization error, with different values of η(the plots and the code refer to the learning rate as 'h'). For the following plots we used a

t_max of 100 (because we have to see when the plots with really low values of η stabilize), numbers of prototypes of 2, 4 and 4 for the data in w6_1x.mat, w6_1y.mat, w6_1z.mat, respectively. A value of η, which makes the plot reasonably stable for all the data sets, seems to be 0.0016, which would thus be a good value to use with the algorithm.

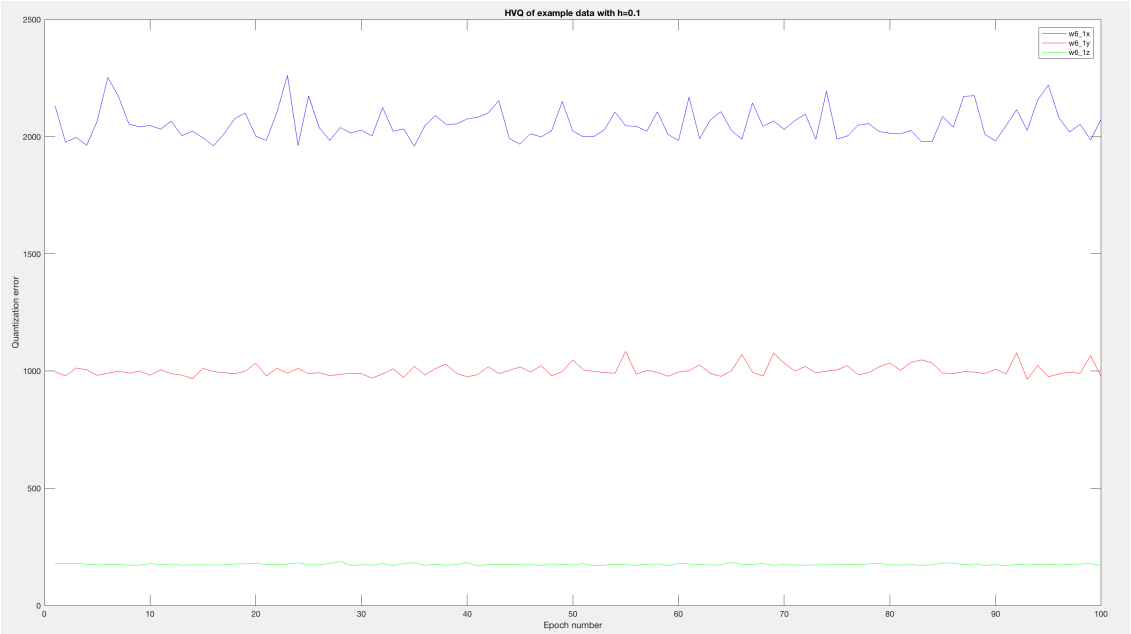

Figure 4: *shows the plot of epoch numbers vs quantization error for all the 3 data sets provided, with a learning rate of 0.1*

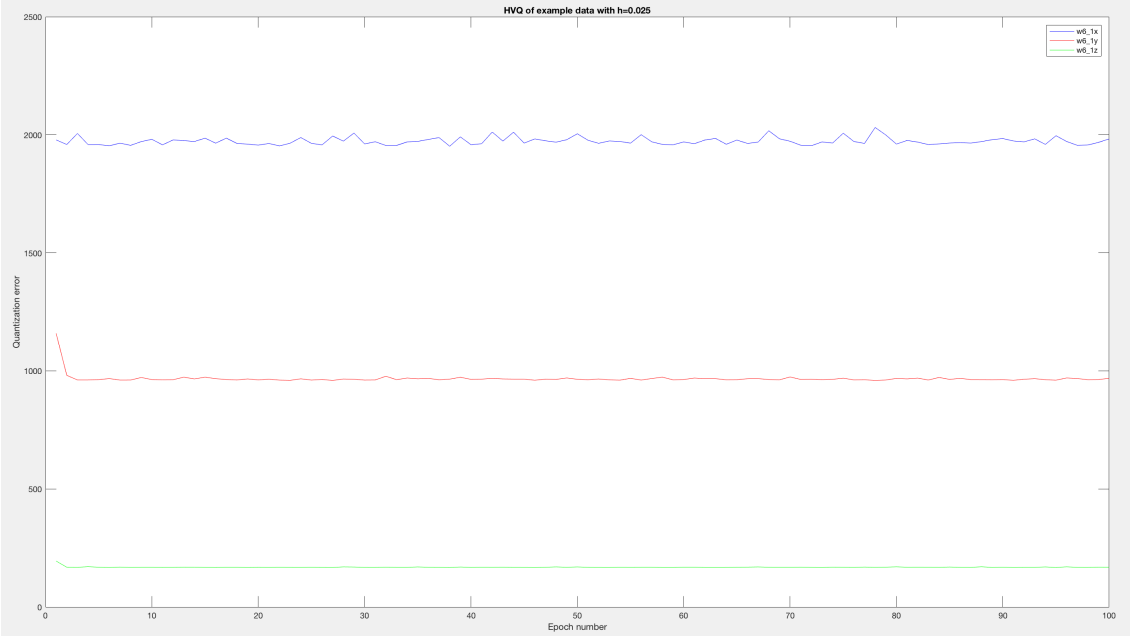

Figure 5: *shows the plot of epoch numbers vs quantization error for all the 3 data sets provided, with a learning rate of 0.025*

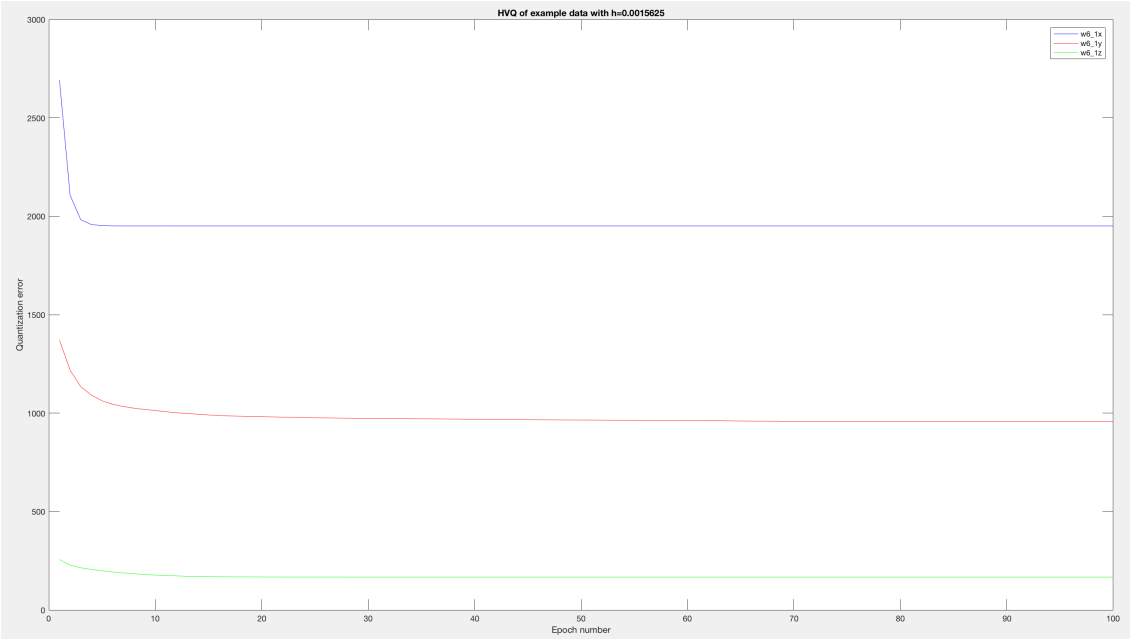

Figure 6: *shows the plot of epoch numbers vs quantization error for all the 3 data sets provided, with a learning rate of 0.0016*

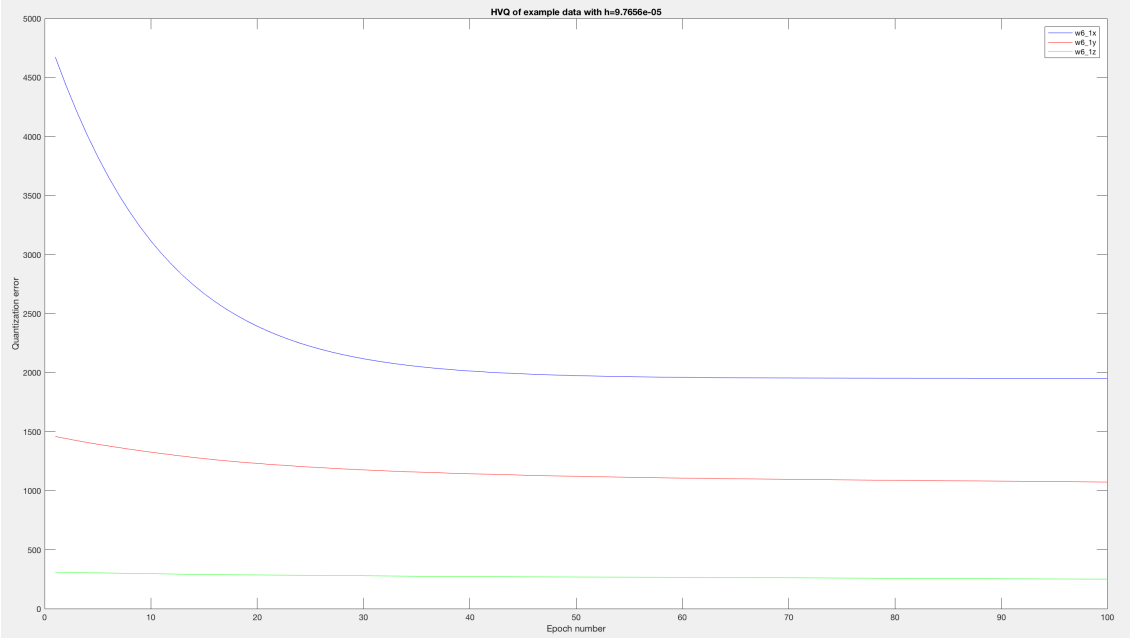

Figure 7: *shows the plot of epoch numbers vs quantization error for all the 3 data sets provided, with a learning rate of* $9.76 * 10^{-5}$

To understand better the effect of the learning rate on the quantization error, we show the plot in **Figure 8** , obtained by plotting the learning rate vs the final quantization error. The figure shows that, with a high value of η the outcome is not predictable (it could be in a local minimum, but we are not sure of that), while with lower values of η we always reach and stay in a minimum (assuming enough epochs are done). For this plot we used the data in w6_1x.mat, with

2 prototypes, and we increased t_max to 300, because otherwise the algorithm would not have reached a stable point for the lowest values of η.

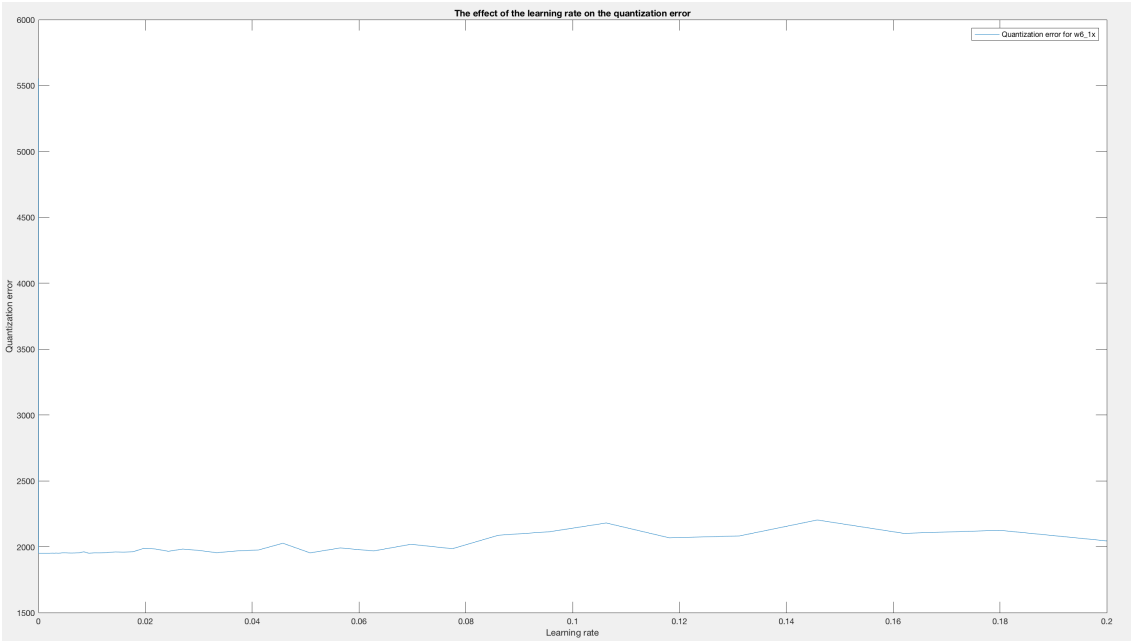

Figure 8: *shows the plot of learning rate vs quantization error for the data set in w6_1x.mat*

However, an optimal value for the learning rate would be a value which decreases slowly after each epoch. If we started with a relatively high learning rate(for instance 0.5), we would manage to get out of the local minima in which we enter, and, by lowering the learning rate in a sufficiently slow way, we should get to the global minimum. To show this we tried the algorithm with a t_max of 1000, an initial learning rate of 0.5, which decreases of 0.1% after each epoch. The plot has been done on each of the given data sets and is shown in **Figure 9** . **Figure 9** shows many of the previously discussed results, such as the high variance of the quantization error when the learning rate is high, and the stable value that it reaches with a lower learning rate.

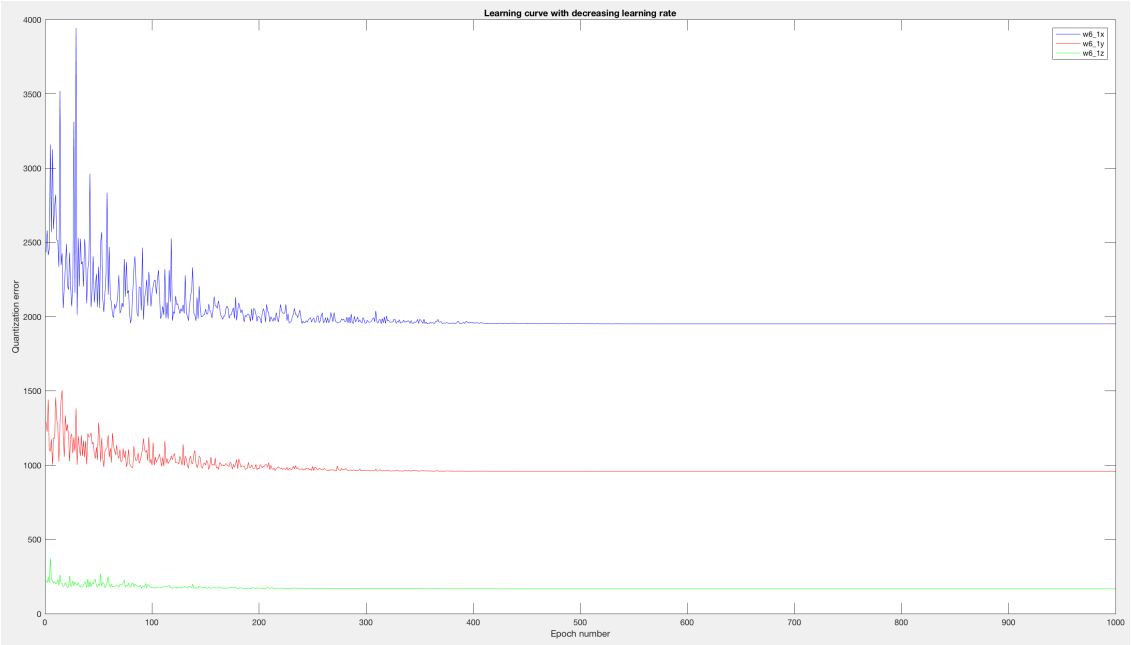

Figure 9: *shows the plot of the learning curve with a decreasing learning rate*

Distance measures

All the plots shown until now have been created by using squared euclidean distance to measure the quantization error. We also tried to use the euclidean distance for the plots, but we noticed that the plots obtained with the two different distance measures had very similar shapes, and were different only for the computed values of quantization error (which is expected, since one sums some values while the other sums their squared values). Thus we decided to use only one of them, and we opted for the squared euclidean distance.

Data distribution

The data distribution plays a central role in the algorithm, as it determines the effect of all the other parameters. This is shown in some of the previous plots, specifically where the plots of the 3 data sets were in the same figure. In those figures, the number of epochs and the learning rate necessary to reach a steady value are significantly different for each data set.

A. Assignment

Listing 1: *This is the file* `script1.m`

```
%Read the input data
Sx = load('w6_1x.mat');
Sy = load('w6_1y.mat');
Sz = load('w6_1z.mat');

Mx = Sx.w6_1x;
My = Sy.w6_1y;
Mz = Sz.w6_1z;

%The structs are not needed anymore
clear Sx Sy Sz;


%Number of prototypes analisys plots
P = size(Mx, 1); h = 0.01; t_max = 100; HVQs = zeros(1, 9);
for K = 1:9
    HVQ = completeLearningProcess(Mx, h, t_max, P, K, 0, 1);
    HVQs(K) = HVQ(t_max);
end
figure(10);
plot(1:9, HVQs);
xlabel('Number of prototypes'); ylabel('Quantization error'); legend('
   Quantization error'); title('Number of prototypes vs Quantization
   error for learning rate=0.01');

%-------------------------------------------------------------

%Learning rate analisys plots

%Plot the learning curve for the 3 data sets for different values of h
   (learning rate)
P = size(Mx, 1); K = [2 4 4]; t_max = 100; h=0.1;
for i = 1:6
    HVQ = completeLearningProcess(Mx, h, t_max, P, K(1), 0, 0);
    figure(10+i);
    p1 = plot(1:t_max, HVQ, 'Color', 'blue');
    fprintf("STD.DEV. for h=%f of last 50 quantization errors: %f\n",h,
   std(HVQ(50:t_max)));

    HVQ = completeLearningProcess(My, h, t_max, P, K(2), 0, 0);
    figure(10+i);
    hold on;
    p2 = plot(1:t_max, HVQ, 'Color', 'red');
    hold off;

    HVQ = completeLearningProcess(Mz, h, t_max, P, K(3), 0, 0);
    figure(10+i);
```

```
    hold on;
    p3 = plot(1:t_max, HVQ, 'Color', 'green');
    legend([p1 p2 p3], 'w6\_1x', 'w6\_1y', 'w6\_1z');

    title(['HVQ of example data with h=' num2str(h)]);
    xlabel('Epoch number');
    ylabel('Quantization error');
    hold off;
    h = h/4;
end

%Plots the learning rate vs the quantization error
h=0.2; learning_rates = zeros(1,100); HVQs= zeros(1,100); t_max = 300;
for i = 1:100
    HVQ = completeLearningProcess(Mx, h, t_max, P, K(1), 0, 0);
    learning_rates(i) = h; HVQs(i) = HVQ(t_max);
    h=h*0.9;
end
figure(17);
plot(learning_rates, HVQs);
legend('Quantization error for w6\_1x');
xlabel('Learning rate');
ylabel('Quantization error');
title('The effect of the learning rate on the quantization error');

%Decreasing learning rate plot
h=0.5; t_max = 1000; P = size(Mx, 1); K = [2 4 4];
HVQ = completeLearningProcess(Mx, h, t_max, P,  K(1), 1, 0);
figure(18);
p1 = plot(1:t_max, HVQ, 'Color', 'blue');
HVQ = completeLearningProcess(My, h, t_max, P, K(2), 1, 0);
figure(18);
hold on;
p2 = plot(1:t_max, HVQ, 'Color', 'red');
hold off;
HVQ = completeLearningProcess(Mz, h, t_max, P, K(2), 1, 0);
figure(18);
hold on;
p3 = plot(1:t_max, HVQ, 'Color', 'green');
legend([p1 p2 p3], 'w6\_1x', 'w6\_1y', 'w6\_1z');
title('Learning curve with decreasing learning rate');
xlabel('Epoch number');
ylabel('Quantization error');
hold off;

%Executes the algorithm on the given parameters, returning the values
   of
%the quantization error after each epoch. The parameter
%update_learning_rate is set to 1 when we want the learning rate to be
%reduced of its 0.1% after each epoch, while we set plot_clusters to 1
    when
```

```
%we want to produce a plot which highlights the different clusters
   with
%different colors
function HVQ = completeLearningProcess(M, h, t_max, P, K,
   update_learning_rate, plot_clusters)
    %Initialize random prototypes from points of the data sets
    prototypes = initializeRandomPrototypes(M, K);

    HVQ = zeros(1, t_max);

    %Clear the figure used to show the algorithm and use it to update
   the
    %situation during the execution
    clf(figure(30));
    figure(30);
    hold on;
    s1 = scatter(M(:,1), M(:,2));
    s2 = scatter(prototypes(:,1), prototypes(:,2), 100, 'filled');
    hold off;
    legend([s1 s2], 'Example data', 'Prototypes');

    for t = 1:t_max
        prototypes = epoch(M, P, K, prototypes, h);
        set(s2,'XData', prototypes(:, 1), 'YData', prototypes(:, 2));
        HVQ(t) = quantizationError(M, prototypes, P, K);
        pause(.01);
        if update_learning_rate == 1
            h = updateLearningRate(h);
        end
    end

    if plot_clusters == 1
        plotClusters(M, prototypes, P, K, t_max, h);
    end
end

%Function called to update the learning rate
function updatedLearningRate = updateLearningRate(h)
    updatedLearningRate = h*0.99;
end

%Function that plots the clusters with different colors on figure(K);
function plotClusters(M, prototypes, P, K, t_max, h)
    group = zeros(1,P);
    colors = [.5 .05 1;1 .1 0; .8 .7 0; .05 .7 1; .55 .25 0; .05 .05 1;
    .95 .05 1; 0 0 0; .3 .5 .7];
    legends = ["Prototype 1"; "Prototype 2"; "Prototype 3"; "Prototype
    4"; "Prototype 5"; "Prototype 6"; "Prototype 7"; "Prototype 8"; "
   Prototype9"];
    for i = 1:P
        [winner, ~] = findWinnerPrototype(M(i,:), prototypes, K);
```

```
        group(i) = winner;
    end
    clf(figure(K));
    figure(K);
    hold on;
    gscatter(M(:,1), M(:,2), group, colors(1:K,:));
    pr = gscatter(prototypes(:,1), prototypes(:,2),1:K, colors(1:K,:),
    '.', 30);
    hold off;
    legend(pr, legends(1:K));
    xlabel("Feature 1");
    ylabel("Feature 2");
    title(['Data clusters with ' num2str(K) ' prototypes, t\_max='
   num2str(t_max) ' and learning\_rate=' num2str(h)]);
end

%Function which computes the quantization error for the given
   prototypes
%and the given data set
function HVQ = quantizationError(M, prototypes, P, K)
    HVQ = 0;
    for i = 1:P
        %Find winner prototype
        [~, winner_distance]  = findWinnerPrototype(M(i,:), prototypes,
    K);

        %Update the quantization error
        HVQ = HVQ + winner_distance;
    end
end

%Function which performs an epoch
function prototypes = epoch(M, P, K, prototypes, h)
    r = randperm(size(M,1)); % permute row numbers
    M = M(r,:);

    %For each example in this dataset
    for i = 1:P
        %Find winner prototype
        [winner, ~] = findWinnerPrototype(M(i,:), prototypes, K);

        %Update winner prototype
        prototypes(winner,:) = prototypes(winner,:) + h*(M(i,:) -
   prototypes(winner,:));
    end
end

%Function which finds the winner prototype and its distance from the
   data
%example v
```

```
function [min_index, min_distance] = findWinnerPrototype(v, prototypes,
    K)
    min_distance = squaredEuclideanDistance(prototypes(1,:), v);
    min_index = 1;
    for k = 2:K
        distance = squaredEuclideanDistance(prototypes(k,:), v);
            if distance < min_distance
                min_distance = distance;
                min_index = k;
            end
    end
end

%Function which computes the squared euclidean distance
function distance = squaredEuclideanDistance(v1, v2)
    distance = (v1(1)-v2(1))^2 + (v1(2) - v2(2))^2;
    %distance = sqrt(distance);
end

%Function which computes the euclidean distance, used only during the
   tests
function distance = euclideanDistance(v1, v2)
    distance = (v1(1)-v2(1))^2 + (v1(2) - v2(2))^2;
    distance = sqrt(distance);
end

%Function that initializes the prototypes randomly as vectors from M
function prototypes = initializeRandomPrototypes(M, K)
    p = ceil(rand(1, K) * size(M,1));
    prototypes = M(p, :);
end
```