Intelligent Systems Assignment 2

Wessel Becker (1982362) & Sander ten Hoor (2318555)

September 27, 2016

1 Matlab Code

The following code was created to obtain the plots.

1.1 main.m

```
load('w6_1x.mat');
load('w6_1y.mat');
load('w6_1z.mat');
x = w6_1x;
y = w6_1y;
z = w6_1z;

dataSets = {x};

numberOfPrototypes = 4;
updateStrat = 'changeDistance';
%learning = 0.004;
learning = 0.001;
tMax = 10;
prototypeStrat = 'randomDataPoints';

for idx = 1:length(dataSets)
    [resultPrototypes, quantizationErrors] =
        ↪ vectorQuantization(idx, dataSets{idx},
        ↪ numberOfPrototypes, updateStrat, learning, tMax
        ↪ , prototypeStrat);
    plotLearningCurve(idx + length(dataSets), tMax,
        ↪ numberOfPrototypes, learning,
        ↪ quantizationErrors);
end
```

1.2 changeDistance.m

```
function [ result ] = changeDistance( selected, target,
    ↪ learning)
%CONSTANTPRODUCT Modify the position of a point, based on
    ↪  another point
```

```
    result = selected + ((target - selected) * learning);
end
```

1.3 randomDataPoints.m

```
function [ chosenPrototypes ] = randomDataPoints( dataset
    ↪ , numberOfPrototypes )
%RANDOMDATAPOINTS Summary of this function goes here
%   Detailed explanation goes here
    chosenPrototypes = datasample(dataset,
        ↪ numberOfPrototypes, 'replace', false);
end
```

1.4 vectorQuantization.m

```
function [ prototypes, quantizationErrors ] =
    ↪ vectorQuantization( setNumber, dataset,
    ↪ numberOfPrototypes, updateStrat, learning, tMax,
    ↪ prototypeStrat )
%VECTORQUANTIZATION Summary of this function goes here
%   Detailed explanation goes here

% Initialize the prototypes by selecting random data
    ↪ points %
% From 1 to tMax: %
%   Shuffle data %
%   for each datapoint, compare to prototypes %
%   Protype that is closest gets updated by n
%   Plot datapoints
%   Evaluate the quantization error Hvq
% Plot the quantization error as a function of t
    prototypes = feval(prototypeStrat, dataset,
        ↪ numberOfPrototypes);
    [numberOfDataPoints, ~] = size(dataset);
    quantizationErrors = zeros(1, tMax);

    for epoch = 1:tMax
        data = datasample(dataset, numberOfDataPoints, '
            ↪ replace', false);

        for idx = 1:numberOfDataPoints
            datapoint = data(idx, :);
            [minimum, distances, selected] = euclidean(
                ↪ datapoint, prototypes);
            updated = feval(updateStrat, selected,
                ↪ datapoint, learning);
            prototypes(distances == minimum, :) = updated
                ↪ ;
        end
        quantizationErrors(epoch) = quantizationError(
            ↪ dataset, prototypes);
```

```
        plotVQ(setNumber, epoch, learning, dataset,
            ↪ prototypes);
    end
end
```

1.5 euclidean.m

```
function [ minimum, distances, selected ] = euclidean(
    ↪ datapoint, prototypes )
%EUCLIDEAN Summary of this function goes here
%   Detailed explanation goes here
    distances = pdist(vertcat(datapoint, prototypes), '
        ↪ euclidean');
    distances = distances(1:length(prototypes));

    minimum = min(distances);
    selected = prototypes(distances == minimum, :);
end
```

1.6 vectorQuantization.m

```
function [ prototypes, quantizationErrors ] =
    ↪ vectorQuantization( setNumber, dataset,
    ↪ numberOfPrototypes, updateStrat, learning, tMax,
    ↪ prototypeStrat )
%VECTORQUANTIZATION Summary of this function goes here
%   Detailed explanation goes here

% Initialize the prototypes by selecting random data
    ↪ points %
% From 1 to tMax: %
%   Shuffle data %
%   for each datapoint, compare to prototypes %
%   Protype that is closest gets updated by n
%   Plot datapoints
%   Evaluate the quantization error Hvq
% Plot the quantization error as a function of t
    prototypes = feval(prototypeStrat, dataset,
        ↪ numberOfPrototypes);
    [numberOfDataPoints, ~] = size(dataset);
    quantizationErrors = zeros(1, tMax);

    for epoch = 1:tMax
        data = datasample(dataset, numberOfDataPoints, '
            ↪ replace', false);

        for idx = 1:numberOfDataPoints
            datapoint = data(idx, :);
            [minimum, distances, selected] = euclidean(
                ↪ datapoint, prototypes);
```

```
            updated = feval(updateStrat, selected,
                ↪ datapoint, learning);
            prototypes(distances == minimum, :) = updated
                ↪ ;
        end
        quantizationErrors(epoch) = quantizationError(
            ↪ dataset, prototypes);
        plotVQ(setNumber, epoch, learning, dataset,
            ↪ prototypes);
    end
end
```

1.7 quantizationError.m

```
function [ quantizationError ] = quantizationError(
    ↪ datapoints, prototypes )
%QUANTIZIATIONERROR Calculate the quantization error for
    ↪ these datapoints
% and the corresponding prototypes

    quantizationError = 0;

    for idx = 1:length(datapoints)
        datapoint = datapoints(idx, :);
        minimum = euclidean(datapoint, prototypes);
        quantizationError = quantizationError + minimum;
    end

end
```

1.8 plotVQ.m

```
function plotVQ( setNumber, epoch, learning, dataset,
    ↪ prototypes )
%PLOTVQ Plot the current state of the quantization vector
    ↪  unsupervised
%learning

    figure(setNumber);
    hold off; plot(0,0); box on;
    axis square; hold on;

    scatter(dataset(:, 1), dataset(:, 2), 300, 'r', '.');
    scatter(prototypes(:, 1), prototypes(:, 2), 400, 'b',
        ↪  '.');
    title(['n = ',num2str(learning,'%f'),          ...
                  ' Epoch = ',num2str(epoch,'%d'),
                    ↪ ...
                  ' Prototypes = ',num2str(length(
                    ↪ prototypes),'%d')],     ...
```

```
                    'fontsize',16);
    set(gca,'fontsize',16);
    xlabel(['X'],'fontsize',16);
    ylabel(['Y'],'fontsize',16);
    set(gca,'fontsize',16)
%     pause(1);
end
```

1.9 plotLearningCurve.m

```
function plotLearningCurve(setNumber, tMax, k, n, results
    ↪ )
%PLOT Plot the quantization error as a function of the
    ↪ epochs

    figure(setNumber);
    hold off; plot(0,0); box on;
    axis square; hold on;

    plot(1:tMax, results);
    title(['n = ',num2str(n,'%f'),          ...
                  ' K = ',num2str(k,'%d')],     ...
                  'fontsize',16);
    set(gca,'fontsize',16);
    xlabel(['Epoch'],'fontsize',16);
    ylabel(['Quantization Error'],'fontsize',16);
    set(gca,'fontsize',16)

end
```

2 Plots

For plotting, dataset x was chosen. The values for the learning rate were 0.1, 0.01 and 0.001. They are included at the end of this document.

3 Discussion

In the discussion the results of, running this simple VQ algorithm for different learning rates and with different numbers of prototypes are discussed. We will focus on the influence of the learning rate on the obtained result but also mention some other interesting observations.

3.1 Fluctuation

At higher learning rates, the quantization error fluctuates a lot more than at lower learning rates, for which the graph are smoother, see figures **??**, **??** and **??** for an example. This effect is caused by the higher movement at higher learning rates. Which make the solutions cause the most recently evaluated data points die be more influential to the chosen values for the prototype vectors. Since the prototypes will change allot towards this points. At lower values for the

learning rate the prototype vectors will evaluate single data points less strongly and therefore tend to stay more related to all data points to which it is closest.

3.2 Number of prototypes

When the number of prototypes is higher, the learning process is slower. This is because every time the distance is evaluated, still only one prototype is updated. If the number of prototypes is twice as large as before, the learning speed for each individual prototype is cut in half.

3.3 Run time versus optimal result

At higher learning rates the algorithm reaches a minimum much faster then at lower learning rates, see for example figures **??** and **??** for a difference. Which means that a good solution could be found in a much smaller time frame. Sadly, this faster learning speed comes at the cost of having a less stable result which could mean a less optimal solution as was discussed in **??**. This means that in practice we will need to make a choice for the learning rate that balances these to factors, keeping in mind the influence of adding more prototypes, see **??**.

3.4 Winner takes all

In one of the plots, there are three prototypes in one half, and one in the other. This affects the quantization error, which might be lower if there were two prototypes on each side. However, this state is never achieved because only the winner is updated. In the right half, the winner is always the sole prototype there. This means the algorithm could easily get stuck in a local minimum.

4 Work done

For this assignment, the algorithm implementation was done by Wessel, while Sander worked on plotting. The latter created the plots and put together this document, after which Wessel made some suggestions and modifications.

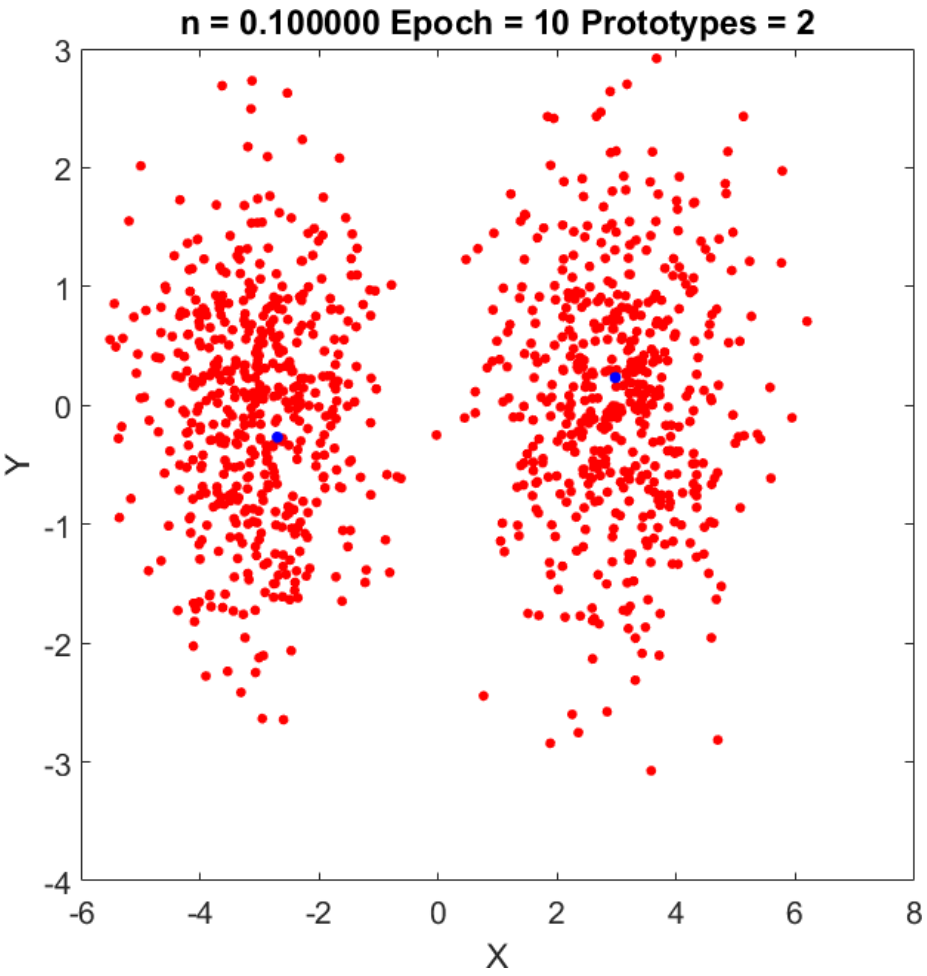

Figure 1: Scatter, n = 0.1, 2 prototypes

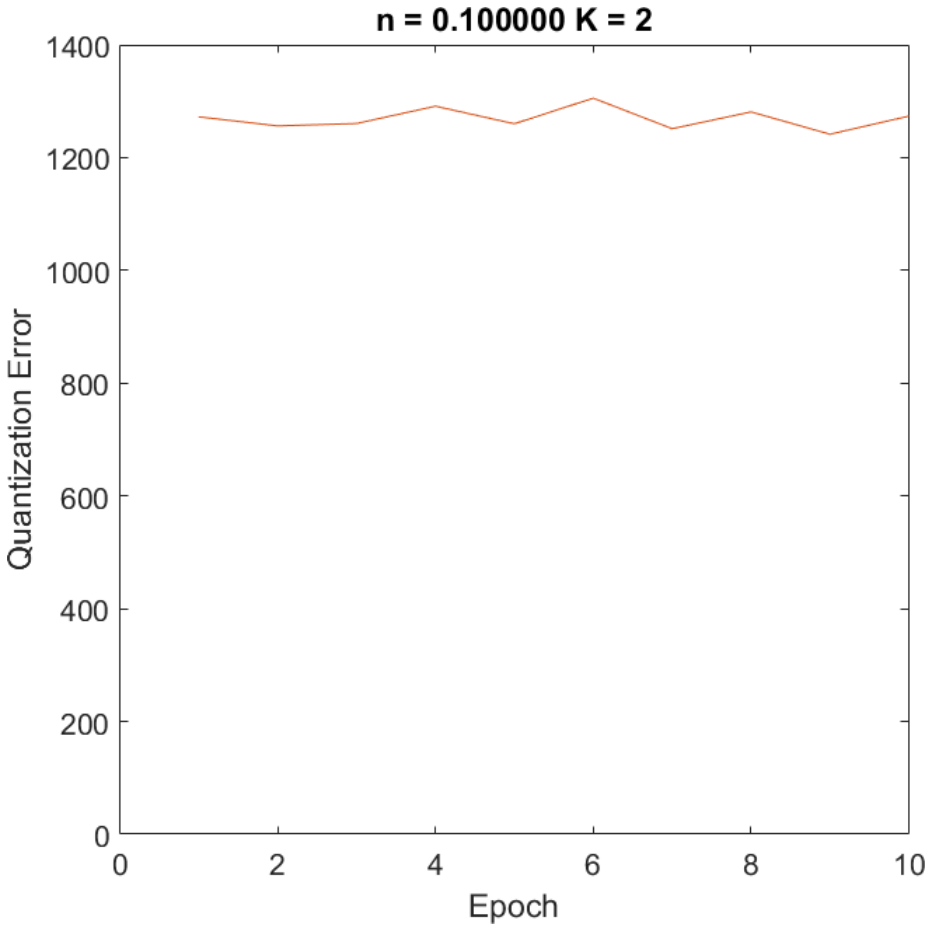

Figure 2: Quantization Error, n = 0.1, 2 prototypes

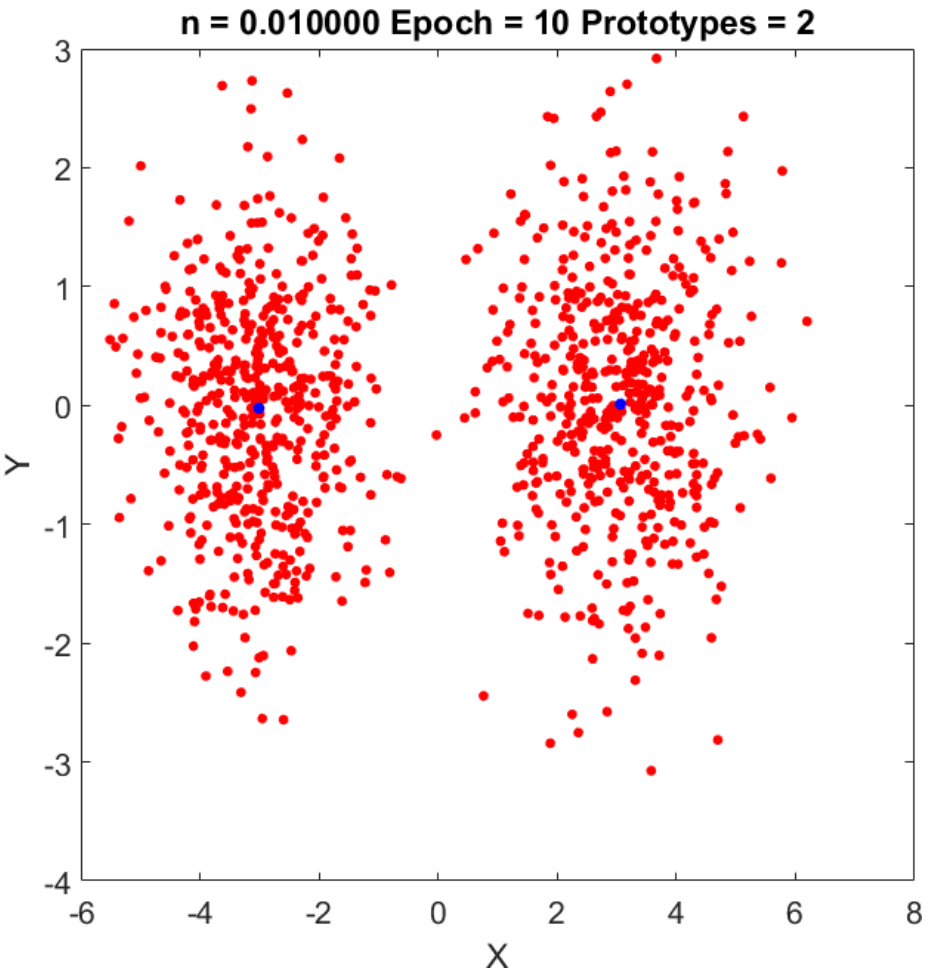

Figure 3: Scatter, n = 0.01, 2 prototypes

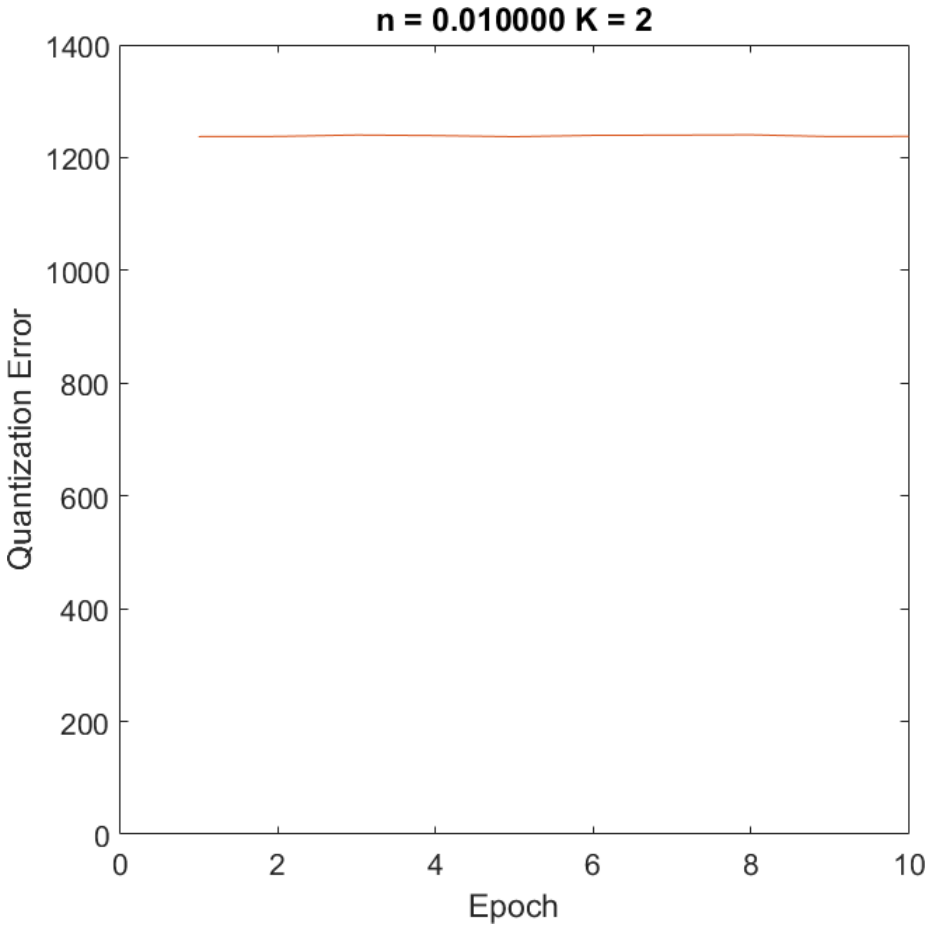

Figure 4: Quantization Error, n = 0.01, 2 prototypes

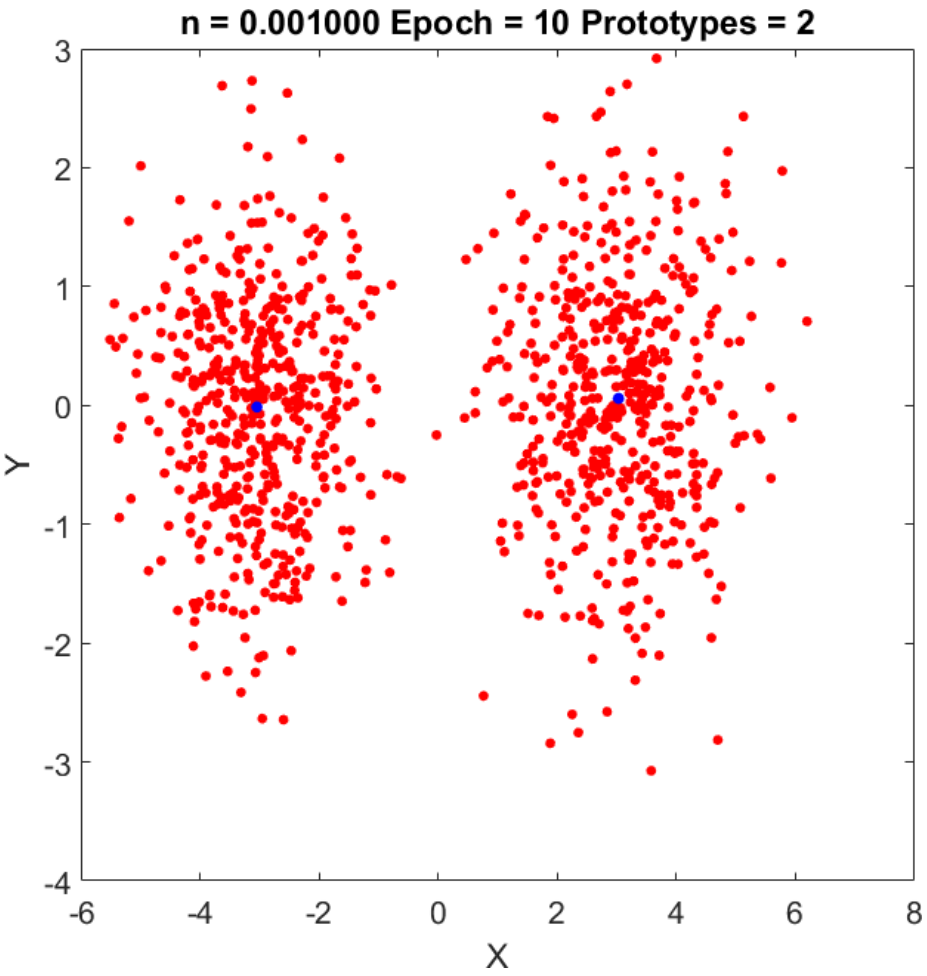

Figure 5: Scatter, n = 0.001, 2 prototypes

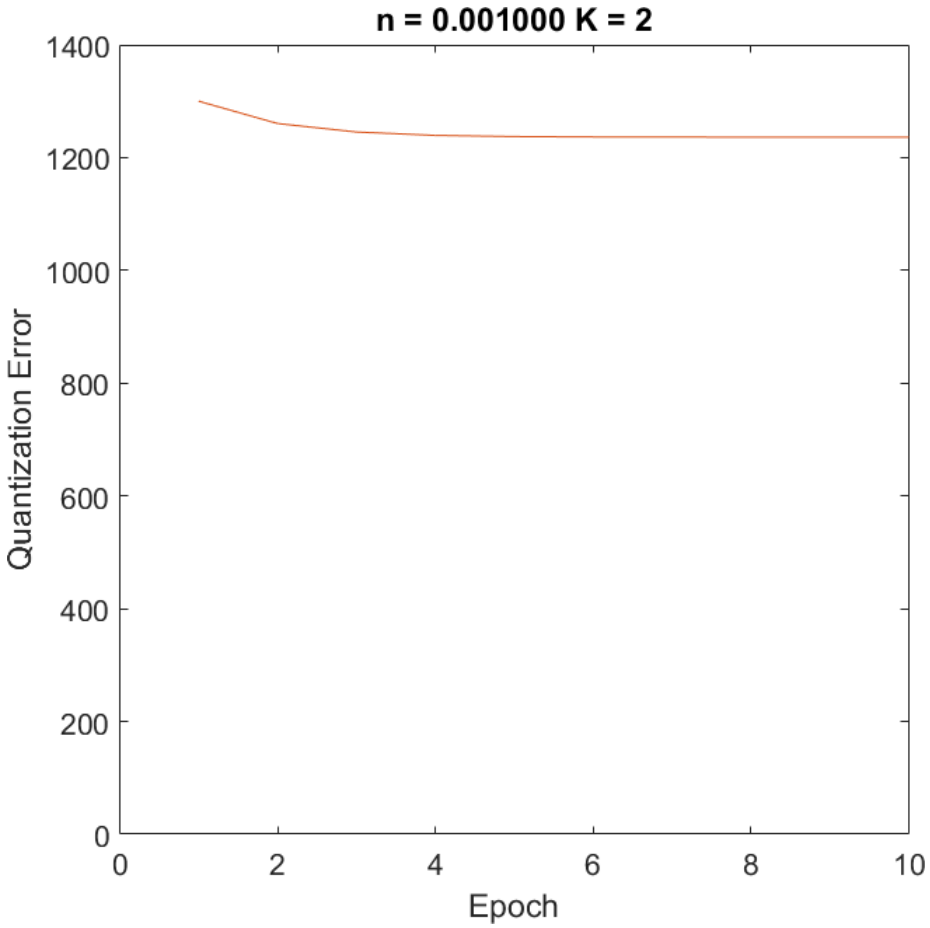

Figure 6: Quantization Error, n = 0.001, 2 prototypes

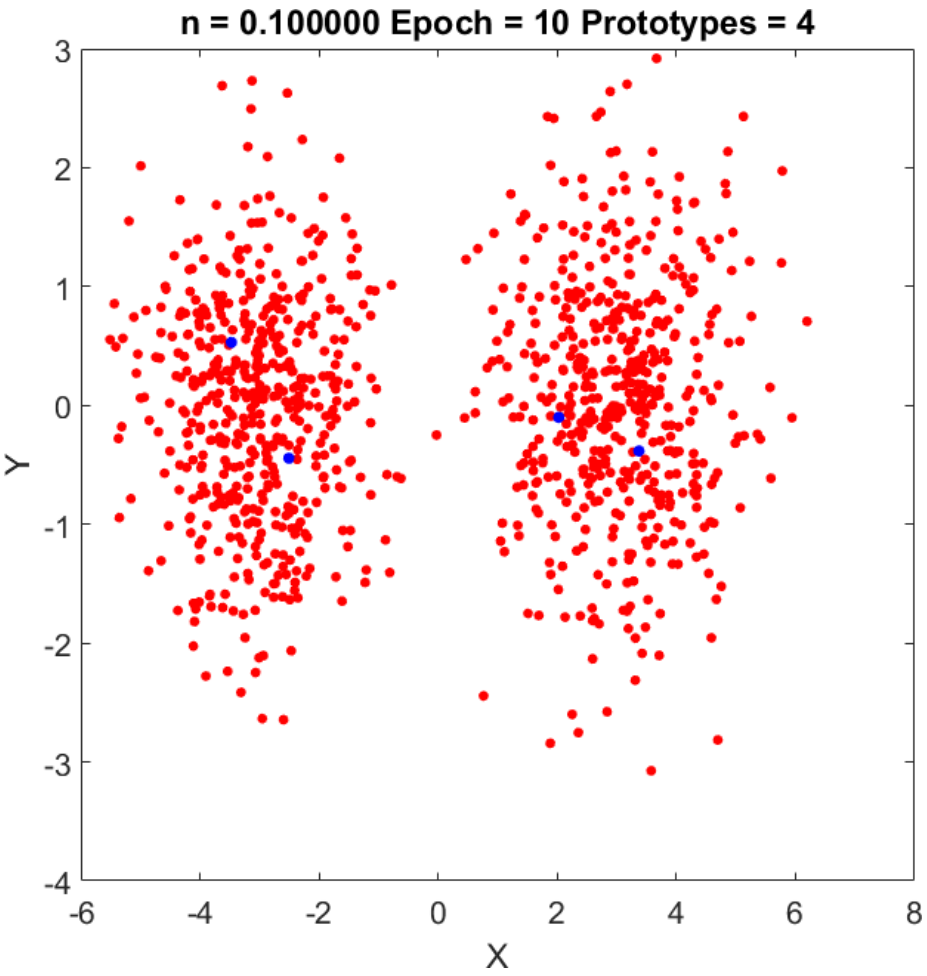

Figure 7: Scatter, n = 0.1, 4 prototypes

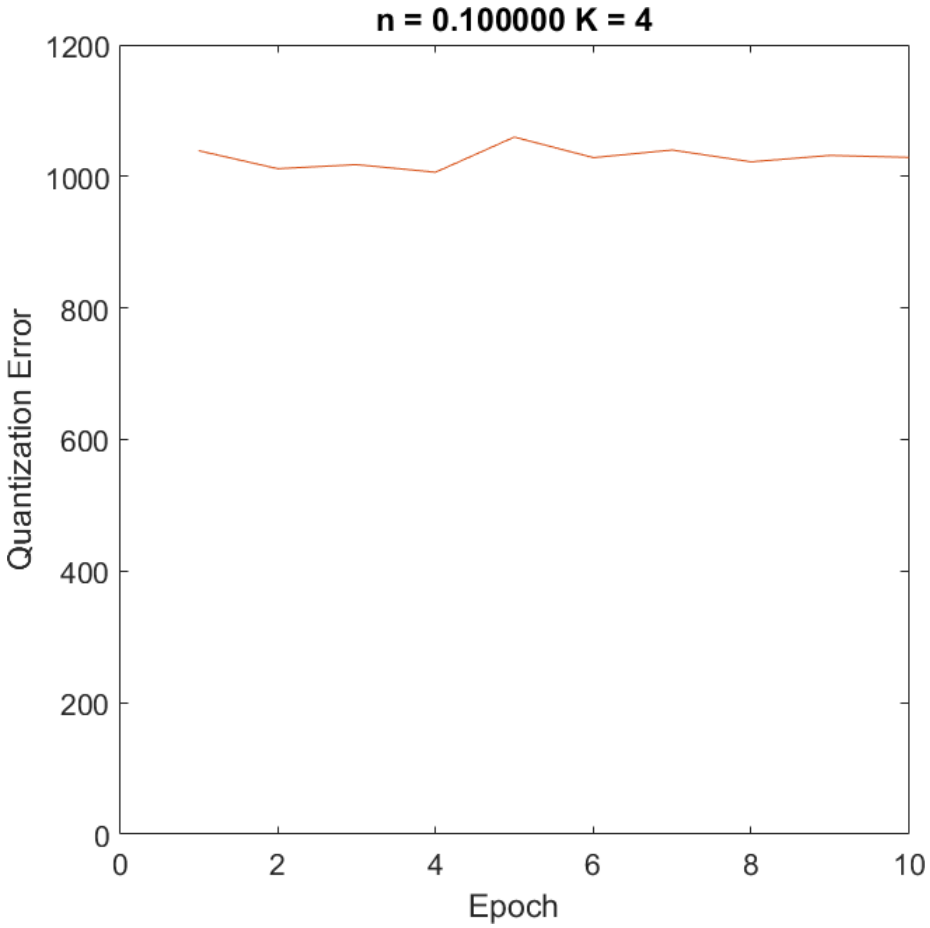

Figure 8: Quantization Error, n = 0.1, 4 prototypes

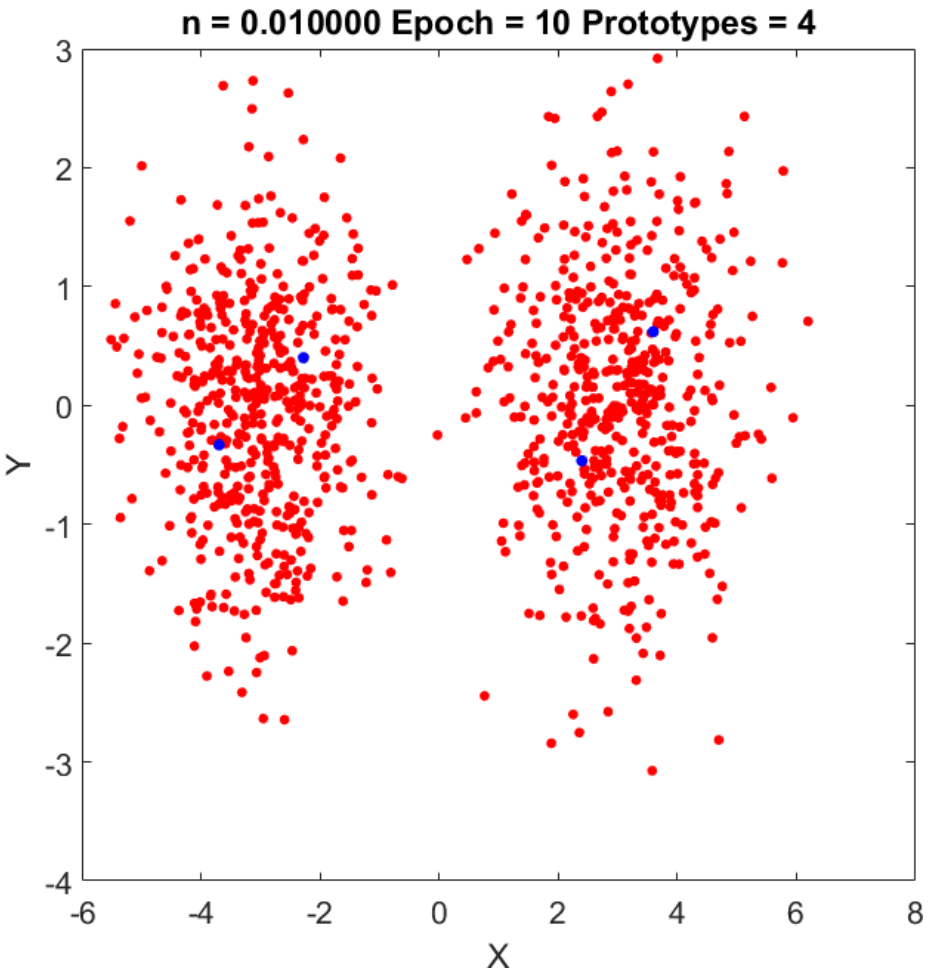

Figure 9: Scatter, n = 0.01, 4 prototypes

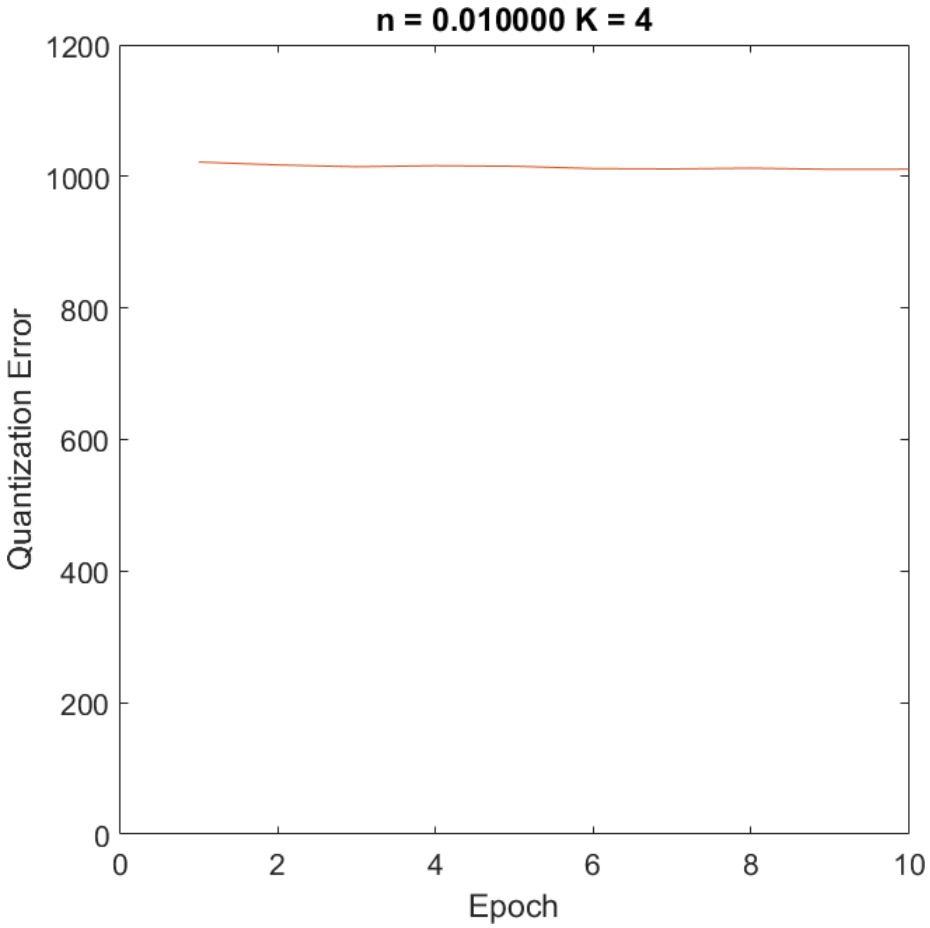

Figure 10: Quantization Error, n = 0.01, 4 prototypes

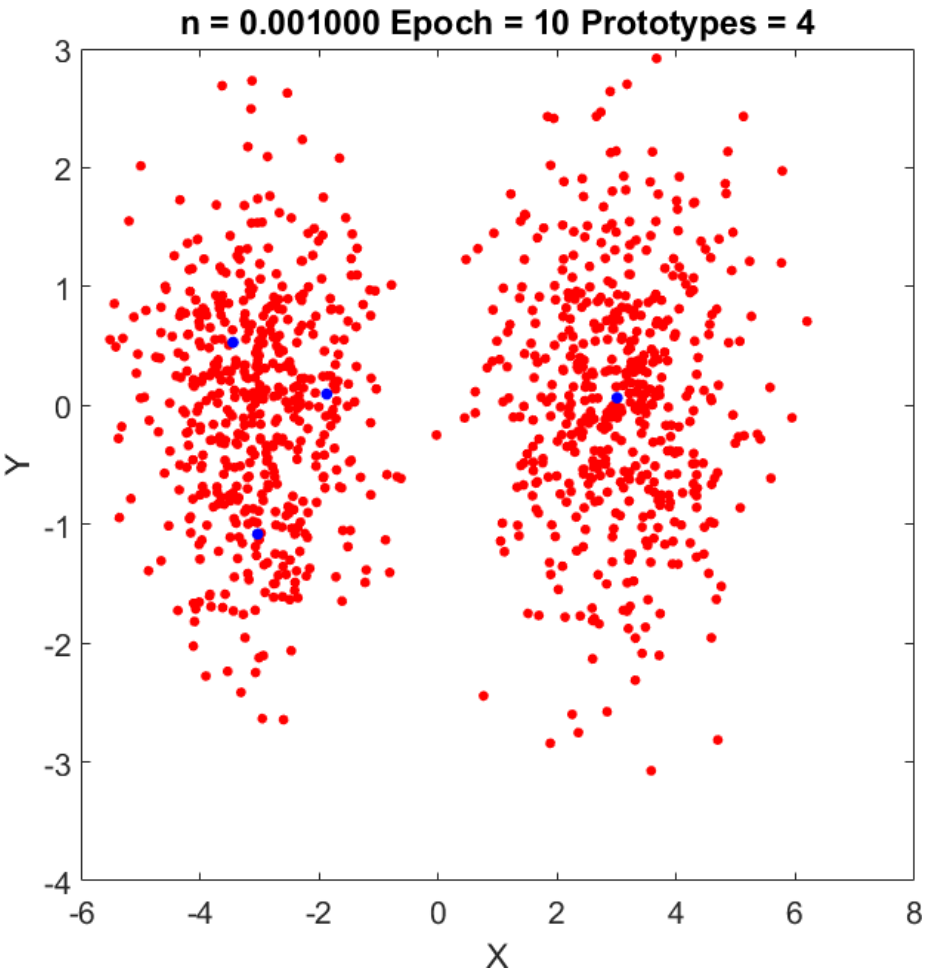

Figure 11: Scatter, n = 0.001, 4 prototypes

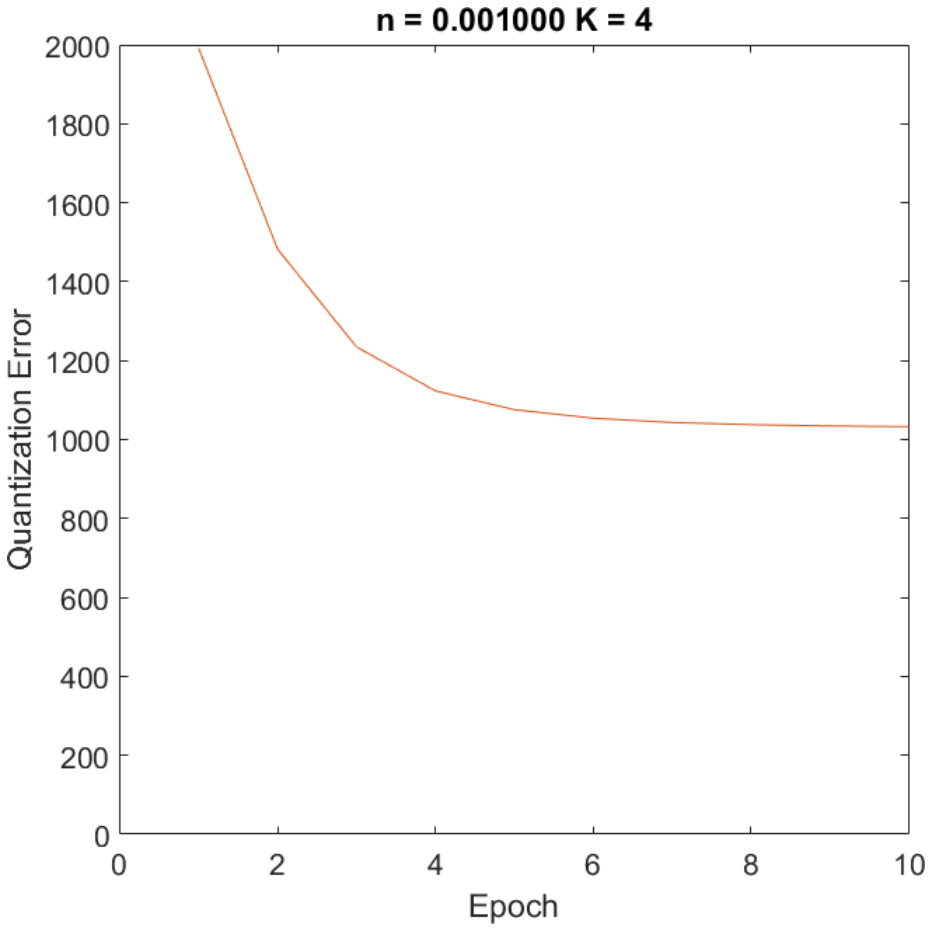

Figure 12: Quantization Error, n = 0.001, 4 prototypes