PSYCH 734 / CSE 734

Assignment 1: Competitive Learning Models

Sara Jamil

Introduction

This report attempts to examine and implement two competitive learning models. The first model uses the Rumelhart and Zipser learning equation with a winner-take-all activation function. The second model uses Kohonen's self-organizing map model which uses the same winner-take-all activation function but with the addition of a neighbourhood function. The neighbourhood function is implemented as a linearly decreasing function such that the network learns in proportion to its distance from the winning unit.

Methods

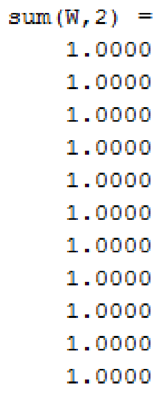
This section describes the models in terms of their respective learning rules and the methods used to test them. Both models use the same initialization function and winner-take-all function. The *Initialize* function sets all the parameters that can be adjusted to different simulations and the *WinnerTakeAll* function computes output activation from the weights and inputs and takes the winner to be the unit with the strongest activation.

Model I – Rumelhart and Zipser's Competitive Learning Model

This competitive learning model described by Rumelhart and Zipser's model must follow a set of properties described below. The details of how they are implemented in the code will also be highlighted.

The units in a given layer are broken into several sets of non-overlapping clusters. Each unit within a cluster inhibits every other unit within a cluster. This is implemented in the winner-take-all function. The output unit receiving the largest input achieves its maximum value of one, while all other units in the cluster are pushed to their minimum value of zero. Therefore, the unit learns if and only if it is the winner of the competition with other units in its layer.

The structure of the network is hierarchical such that every unit in the layer receives inputs from all members of the same set of input units. The input units consist of a set of binary patterns on which the model performs training for a specified number of repetitions. Each of these connections between the input and output layer has a strength given as a weighting which must be a positive value. The total weight from all input units to each output unit must equal 1. When the weight matrix is initialized to a random set of numbers, it is normalized to ensure this property holds, and it is maintained by the learning rule that specifies how the weights will update. The following summation shows that this property holds in the implementation.



```
sum(W,2) =
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

Figure 1: Summation of inputs to output units

A unit learns by shifting weight from its inactive to its active input lines. If a unit wins the competition, then each of its input lines gives up some portion ϵ of its weight and that weight is then distributed equally among the active input lines. Otherwise, if the unit does not win the competition, its weights do not get updated and it does not undergo learning. The learning rule is given by the following equation:

$$\Delta w_{ij} = \begin{cases} 0 & \text{if unit } i \text{ loses on stimulus } k \\ \epsilon \frac{active_{jk}}{nactive_k} - \epsilon w_{ij} & \text{if unit } i \text{ wins on stimulus } k \end{cases}$$

Figure 2: Competitive Learning Rule

active$_{jk}$ is equal to 1 if in stimulus pattern S_k
unit *j* in the lower layer is active and is zero otherwise
nactive$_k$ is the number of active units in pattern S_k

In the Matlab implementation, the learning equation is performed using the following code. This learning rule is repeated for each of the input patterns and the process is repeated overall for a specified number of repetitions in order for the weights to converge to a stable output.

```
InputAC = InputPatterns(:,p);
OutputAC = WinnerTakeAll(W, InputAC); % apply Winner-Take-All

%%% apply Learning Rule
% unit learns iff it wins the competition
deltaWwinner = g*(InputAC'./sum(InputAC) - OutputAC'*W);
deltaW = OutputAC*deltaWwinner;
W = W + deltaW; % update the weights
```

The weight updated is implemented in matrix form where the only non-zero values of deltaW are in the row of the winning output. The difference between the Matlab implementation and the learning rule shown above is that the *nactive* variable which is the number of active units in the pattern is implemented as a sum of the inputs. They are equivalent if the output is assumed to be binary but will differ when non-binary inputs are given (this is discussed in a later section).

After the model is trained for the given number of repetitions, the model is tested using the *WinnerTakeAll* function. It takes the final weight matrix and multiplies it by the specified input pattern.

```
for i = 1:Npats
    test(:,i) = WinnerTakeAll(W, P(:,i));
end
```

The code above is implemented in the *testRumelhart* function and tests the output for the given input patterns. It can also be used for any specified input that may differ from the input patterns.

Model II – Kohonen's Self-Organizing Map Model

The Kohonen network model shares many of the same properties as the model described above. However, the key difference is the addition of a neighbourhood function that allows neighbouring units to the winning output undergo learning as well but to a lesser extent.

The neighbourhood function is useful for attempting to cluster similar types of inputs. It is modelled as a linearly decreasing function with respect to distance of the unit from the winning unit. In the Matlab implementation, the only additional input required is the slope, *m*, of the function given as a value between 0 and 1. The following is the neighbourhood function implementation in the *trainKohonen* function.

```
%%% neighbourhood function
% decreases linearly with distance from the winner
i = find(OutputAC>0);
neigh = zeros(size(OutputAC));
for x = 1:length(OutputAC)
    if(x<=i)
        neigh(x) = m*(x-i)+1;
    else
        neigh(x) = -m*(x-i)+1;
    end
end
neigh(neigh<0) = 0;
```

The following plots show what the neighbourhood function would look like if the winning unit is the 5th unit in a total of 11 outputs. The vector created will have a maximum value of 1 at the winning output and the weights will decrease linearly with respect to distance and will not take on values less than 0.

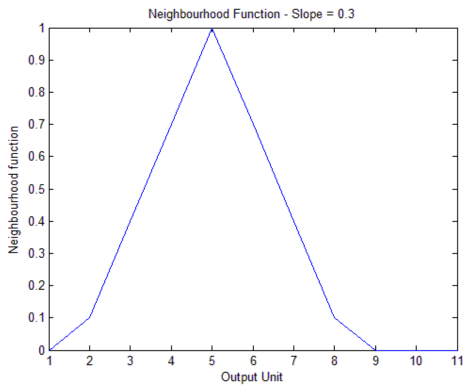
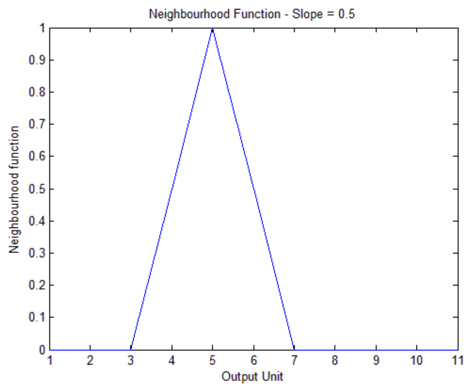



Figure 3: Neighbourhood function vector of slope 0.3 and 0.5

The learning rule applied for this model is very similar to that of the previous model with the additional multiplication of the neighbourhood value for the specific output unit.

```
%%% apply Learning Rule
% same learning rule with addition of neighbourhood function
deltaW = zeros(size(W));
for k = 1:Noutputs
    deltaW(k,:) = g*neigh(k)*(InputAC'./sum(InputAC) - W(k,:));
    W(k,:) = W(k,:) + deltaW(k,:); % update the weights
end
```

The method used for testing inputs is the same as that used for testing the first model (i.e. using the *WinnerTakeAll* function).

Results

This section illustrates the results of the two models with its given input parameters. Each time the simulation is run, a new random weight matrix is initialized, so the exact values of the results will change but the general properties of the output remain the same.

Model I – Rumelhart and Zipser's Competitive Learning Model

The results obtained from the Rumelhart and Zipser model are shown below for a few different starting randomized initial weights. Each time a simulation is run different outputs are produced with seemingly no structure to how the outputs are arranged. Also, the number of unique outputs may change between trials. The following figures show two different simulations where 5 and 4 unique outputs are shown, respectively. The number of output units was chosen to be the same as the number of input patterns to see whether the model would be able to differentiate between all the different input patterns. The columns of *test* represent a single output for the input patterns in the same order they were given.

```
test =

     0     0     0     0     0     0     0     0     0     0     0
     0     0     1     1     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0     0
     1     1     0     0     0     0     0     0     0     0     0
     0     0     0     0     1     1     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     1     1
```

Figure 4: Example test 1 – Model I

```
test =

     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     1     1
     0     0     1     1     1     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     1     1     0     0     0     0     0     0     0     0     0
```

Figure 5: Example test 2 – Model II

There does not seem to be any structure between neighbouring input patterns, as expected. The output also depends on the value of the learning rate and the number of repetitions. However, testing different values of these parameters produce the same basic properties of the outputs. The value of the learning rate used in these examples was 0.1. The weights were randomly tested to ensure that they converge on a value after the specified number of repetitions. The example below clearly shows that the weights do converge to a value and become stable before the end of its training.

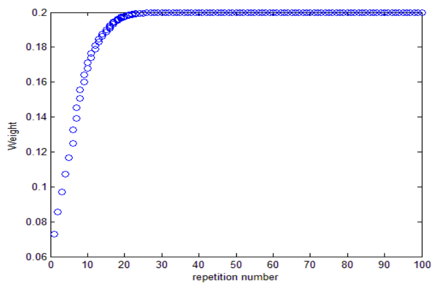
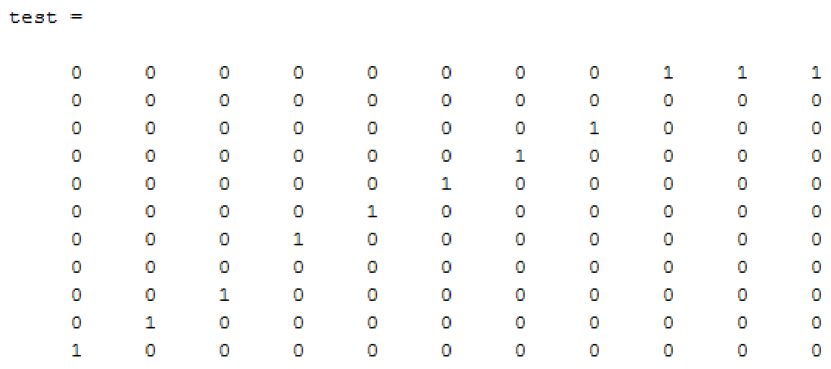


Figure 6: Convergence of weights

Model II – Kohonen's Self-Organizing Map Model

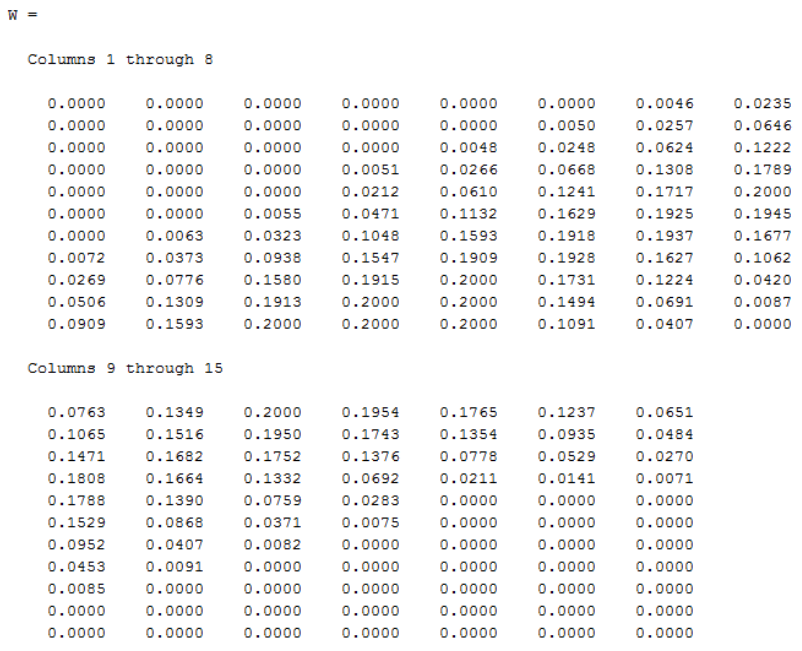
The results demonstrated from the Kohonen Network model show clear differences from the previous model and have a structure to the outputs with respect to the input patterns. Similar input patterns activate neighbouring output units as expected from the use of the neighbourhood function. Also, there appear to be more unique outputs present with respect to input patterns. In the example below, the model is able to differentiate between 9 input patterns. The slope used in this example was 0.3. Although not all 11 of the input patterns were differentiated, more unique outputs were produced than the previous model.



```
test =

     0     0     0     0     0     0     0     0     1     1     1
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     1     0     0     0
     0     0     0     0     0     0     1     0     0     0     0
     0     0     0     0     0     1     0     0     0     0     0
     0     0     0     0     1     0     0     0     0     0     0
     0     0     0     1     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0     0     0
     0     1     0     0     0     0     0     0     0     0     0
     1     0     0     0     0     0     0     0     0     0     0
```

Figure 7: Example test - Model II

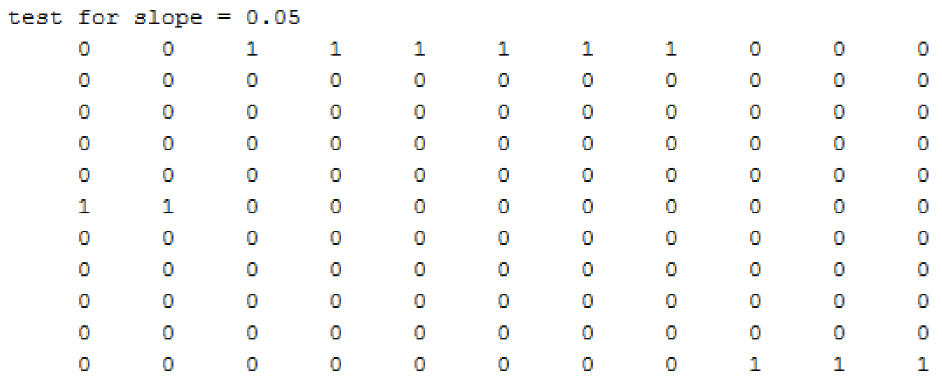
The weight matrix produced in this example is shown below. This helps to illustrate why perhaps not all of the patterns are distinguishable. The weightings display the same structure but the values for neighbouring input patterns are very similar and may be hard to distinguish.



```
W =

  Columns 1 through 8

    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0046    0.0235
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0050    0.0257    0.0646
    0.0000    0.0000    0.0000    0.0000    0.0048    0.0248    0.0624    0.1222
    0.0000    0.0000    0.0000    0.0051    0.0266    0.0668    0.1308    0.1789
    0.0000    0.0000    0.0000    0.0212    0.0610    0.1241    0.1717    0.2000
    0.0000    0.0000    0.0055    0.0471    0.1132    0.1629    0.1925    0.1945
    0.0000    0.0063    0.0323    0.1048    0.1593    0.1918    0.1937    0.1677
    0.0072    0.0373    0.0938    0.1547    0.1909    0.1928    0.1627    0.1062
    0.0269    0.0776    0.1580    0.1915    0.2000    0.1731    0.1224    0.0420
    0.0506    0.1309    0.1913    0.2000    0.2000    0.1494    0.0691    0.0087
    0.0909    0.1593    0.2000    0.2000    0.2000    0.1091    0.0407    0.0000

  Columns 9 through 15

    0.0763    0.1349    0.2000    0.1954    0.1765    0.1237    0.0651
    0.1065    0.1516    0.1950    0.1743    0.1354    0.0935    0.0484
    0.1471    0.1682    0.1752    0.1376    0.0778    0.0529    0.0270
    0.1808    0.1664    0.1332    0.0692    0.0211    0.0141    0.0071
    0.1788    0.1390    0.0759    0.0283    0.0000    0.0000    0.0000
    0.1529    0.0868    0.0371    0.0075    0.0000    0.0000    0.0000
    0.0952    0.0407    0.0082    0.0000    0.0000    0.0000    0.0000
    0.0453    0.0091    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0085    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
```

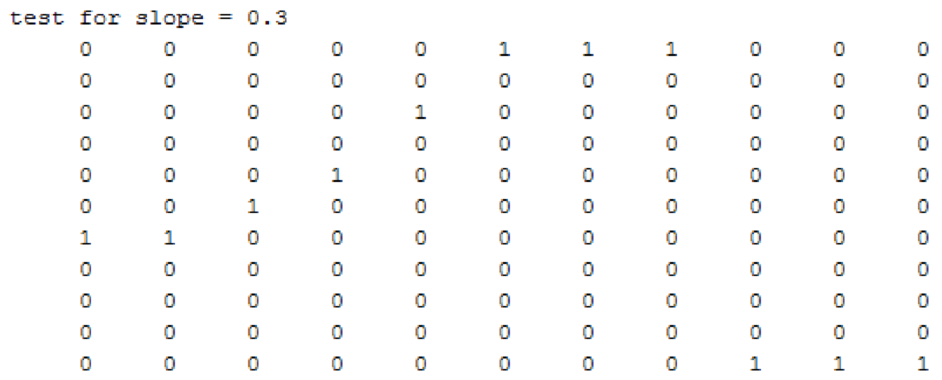
Figure 8: Weights for test

The performance of the model also depends on the value of the slope for the neighbourhood function. A few test cases for different slopes are shown below. When the slope is very small and the neighbourhood function activates most of the neighbouring units to a relatively high degree, there are very few different unique outputs presented. In this test case where slope = 0.05, only three unique outputs are present. In general, it seems that larger slopes perform better at producing more unique outputs. However, if the slope is equal to 1, this model will be equivalent to the Rumelhart and Zipser model.



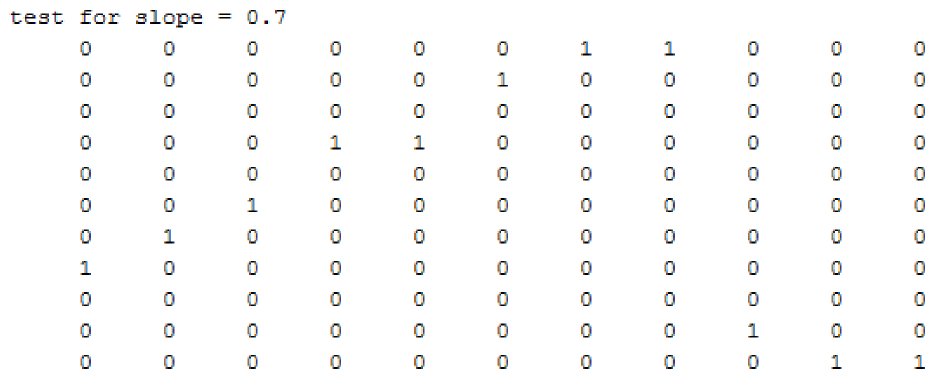
```
test for slope = 0.05
     0     0     1     1     1     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     1     1     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     1     1     1
```

Figure 9: m = 0.05



```
test for slope = 0.3
     0     0     0     0     0     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     1     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     1     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0     0     0
     1     1     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     1     1     1
```

Figure 10: m = 0.3



```
test for slope = 0.7
     0     0     0     0     0     0     1     1     0     0     0
     0     0     0     0     0     1     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     1     1     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0     0     0
     0     1     0     0     0     0     0     0     0     0     0
     1     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     1     0     0
     0     0     0     0     0     0     0     0     0     1     1
```

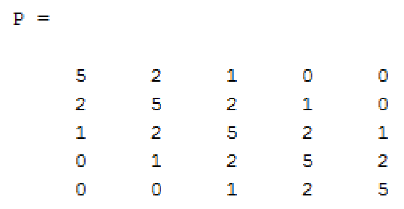
Figure 11: m = 0.7

Discussion

In comparing the two models, some clear differences appear as illustrated in the results section above. The Rumelhart and Zipser model did not have any structure to the outputs with respect to the input patterns. The Kohonen model also was able to produce more unique outputs as compared to the first model. The neighbourhood function allows it to find structure in the input patterns, and for certain values of the slope of the function, it was able to outperform the first model in the number of inputs it is able to distinguish between. The Kohonen model is also likely to be a better model of the brain since the neighbourhood function allows it to model the connections that occur between neurons in the same level in the cortex.

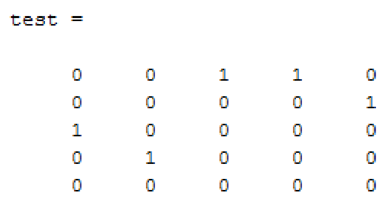
Inputting a pattern of all 1's or all 0's presents a special case in the models. An input of all 1's causes all of the outputs to equal 1, so there is no clear winner in the winner-takes-all scenario. Similarly, an input of all 0's sets all outputs to 0, so again no clear winner arises. These are edge cases that occur only at extreme operating parameters. The output in these cases may become the first or last unit by default and would need special handling in the algorithm.

If the model were to be trained on real-valued patterns rather than binary patterns it would need modification. The learning rule requires dividing $active_{jk}$ by $nactive_{k}$, where $nactive_{k}$ is the number of active units. In a real-valued problem, all of the units may be active to different extents, so using this formula exactly wouldn't be applicable. However, in implementing the equation as shown in the Model I Methods section with directly taking the input values and dividing by the sum of the inputs, the adjusted model is able to take in positive, real-valued problems. An example of non-binary input that was tested is shown below. The input patterns are shown in *P*, the output of the Rumelhart and Zipser model is shown in the middle, and the output of the Kohonen model is shown on the right. This modification of the model still ensures that the sum of the inputs to each output unit is equal to 1. In the following example, the number of repetitions used was 100, the learning rate was 0.1, and the slope of the neighbourhood function for the Kohonen model was 0.8. Their performance seems to agree with the behaviours observed in the Results section above, so this may suggest a possible way to adjust the model to deal with positive, real-valued problems.



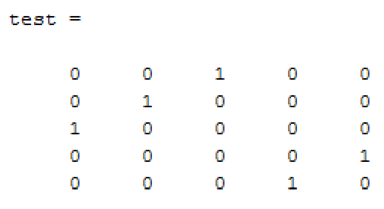
```
P =

     5     2     1     0     0
     2     5     2     1     0
     1     2     5     2     1
     0     1     2     5     2
     0     0     1     2     5
```

Figure 12: Real-valued input



```
test =

     0     0     1     1     0
     0     0     0     0     1
     1     0     0     0     0
     0     1     0     0     0
     0     0     0     0     0
```

Figure 13: Rumelhart & Zipser



```
test =

     0     0     1     0     0
     0     1     0     0     0
     1     0     0     0     0
     0     0     0     0     1
     0     0     0     1     0
```

Figure 14: Kohonen

References

McClelland, James L., and David E. Rumelhart. *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. Cambridge, MA: MIT, 1988. Print.

Appendix A – *Initialize.m*

```
%%%%%%%%% Initialize
% A script that sets global variables, e.g. layer sizes, learning rates,
% initial weights, training patterns. This will be called once from the
% command line for each simulated model that you run.

% training patterns (input data)
P = zeros(15,11);
for i = 1:11
    P(i:i+4,i) = 1;
end

Npats = size(P,2); %number of training patterns
Ninputs = size(P,1); %number of input units
Noutputs = 11;
m = 0.8; %slope of linear neighbourhood function (Kohonen)

Nreps = 100; %number of repetitions (of P)
g = 0.1; %learning rate

W = rand(Noutputs, Ninputs); %random initialization of Weights

for i = 1:Noutputs
    W(i,:) = W(i,:)/sum(W(i,:)); %normalize to have sum(row)=1
end

% % testing algorithm with non-binary inputs
% P = [5 2 1 0 0;
%     2 5 2 1 0;
%     1 2 5 2 1;
%     0 1 2 5 2;
%     0 0 1 2 5]';
%
% Noutputs = 5;
```

Appendix B – *WinnerTakeAll.m*

```
function [outputActivations] = WinnerTakeAll(WeightMatrix, InputPattern)
% Winner-take-all
% max() takes first instance of max value
% set max index = 1, otherwise = 0

Noutputs = size(WeightMatrix,1);
netInput = WeightMatrix*InputPattern;

[out, index] = max(netInput);
outputActivations = zeros(Noutputs, 1);
outputActivations(index) = 1;

end
```

Appendix C – *trainRumelhart.m*

```
function [newWeights] = trainRumelhart(oldWeights, InputPatterns, learningRate, numIterations)
% Training function
% Rumelhart & Zipser's
% Competitive Learning Model

W = oldWeights;
g = learningRate;
Nreps = numIterations;
Npats = size(InputPatterns,2);

for rep = 1:Nreps
```

```
    for p = 1:Npats
        InputAC = InputPatterns(:,p);
        OutputAC = WinnerTakeAll(W, InputAC); % apply Winner-Take-All

        %%% apply Learning Rule
        % unit learns iff it wins the competition
        deltaWwinner = g*(InputAC'./sum(InputAC) - OutputAC'*W);
        deltaW = OutputAC*deltaWwinner;
        W = W + deltaW; % update the weights
    end
end

newWeights = W;

end
```

Appendix D – *testRumelhart.m*

```
% Rumelhart & Zipser's
% Competitive Learning Model
% Model 1
clear all; close all; clc;

Initialize

W = trainRumelhart(W, P, g, Nreps);

% to test an input, use WinnerTakeAll
for i = 1:Npats
    test(:,i) = WinnerTakeAll(W, P(:,i));
end
test
```

Appendix E – *trainKohonen.m*

```
function [newWeights] = trainKohonen(oldWeights, InputPatterns, slope, learningRate,
numIterations)
% Training function
% Kohonen's Model
% Self-organizing map (SOM)
%
% (includes input for slope of neighbourhood function)

W = oldWeights;
g = learningRate;
Nreps = numIterations;
Npats = size(InputPatterns,2);
m = slope;
Noutputs = size(W,1);

for rep = 1:Nreps
    for p = 1:Npats
        InputAC = InputPatterns(:,p);
        OutputAC = WinnerTakeAll(W, InputAC); % apply Winner-Take-All

        %%% neighbourhood function
        % decreases linearly with distance from the winner
        i = find(OutputAC>0);
        neigh = zeros(size(OutputAC));
        for x = 1:length(OutputAC)
            if(x<=i)
                neigh(x) = m*(x-i)+1;
            else
                neigh(x) = -m*(x-i)+1;
            end
        end
        neigh(neigh<0) = 0;

        %%% apply Learning Rule
        % same learning rule with addition of neighbourhood function
        deltaW = zeros(size(W));
        for k = 1:Noutputs
            deltaW(k,:) = g*neigh(k)*(InputAC'./sum(InputAC) - W(k,:));
            W(k,:) = W(k,:) + deltaW(k,:); % update the weights
        end
    end
end

newWeights = W;

end
```

Appendix F – *testKohonen.m*

```
% Kohonen's
% Self Organizing Map
% Model 2
clear all; close all; clc;

Initialize

W = trainKohonen(W, P, m, g, Nreps);

test = zeros(Noutputs,Npats);
for i = 1:Npats
    test(:,i) = WinnerTakeAll(W, P(:,i));
end
test

% % testing with respect to slope of neighbourhood function
% for m = 0.05:0.05:0.9
%     W = trainKohonen(W, P, m, g, Nreps);
%     test = zeros(Noutputs,Npats);
%     for i = 1:Npats
%         test(:,i) = WinnerTakeAll(W, P(:,i));
%     end
%     disp(['test for slope = ', num2str(m)]); disp(test);
% end
```