Sergiu Iliev | Bayesian ML Course | Final | Problem 3

Restricted Boltzmann Machines: Labelling Street View House Numbers (SVHN)

```
clc, clear
```

Data Preparation

(a) Loading the dataset, converting to grayscale and re-shaping

Importing Street Numbers Images Dataset

```
train_32x32 = load('train_32x32.mat');  % load training dataset
test_32x32 = load('test_32x32.mat');    % load testing dataset
```

Visualize the training dataset

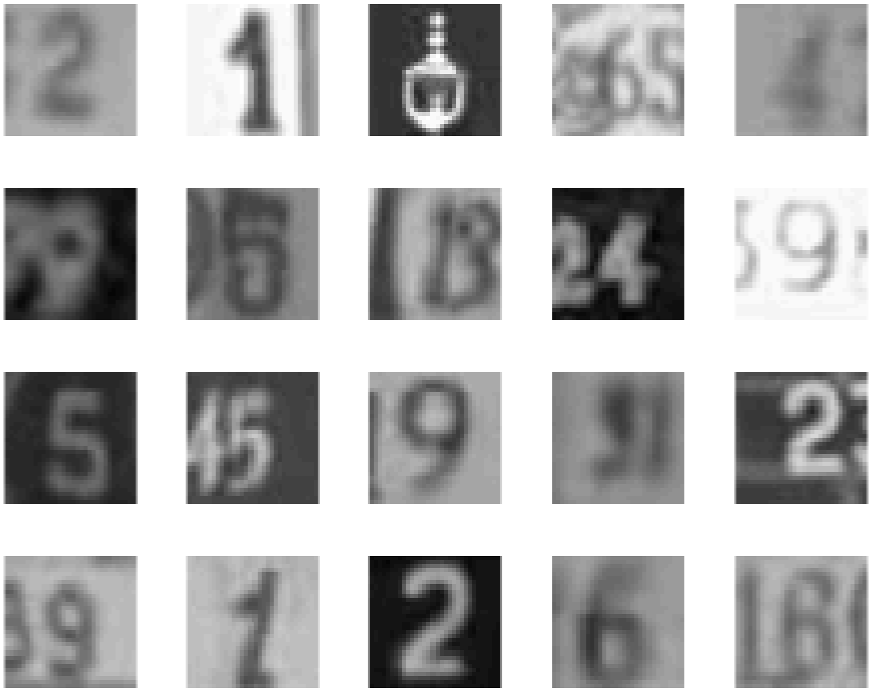
```
figure;
test_slates_no = size(test_32x32.X, 4);     % find the number of image slates in the test datas
perm = randperm(test_slates_no,20);         % create an array of 20 random numbers
for i = 1:20                                % plot 20 random images from the dataset
    subplot(4,5,i);
    imshow(test_32x32.X(:,:,:,perm(i)));
end
```

Convert the images to grayscale

```
test_img_no = size(test_32x32.X, 4);                 % Find the number of images in the test da
for i = 1:test_img_no                                % loop over all test images to convert the
    test_32x32_grey.X(:,:,:,i) = rgb2gray(test_32x32.X(:,:,:,i));
end

train_img_no = size(train_32x32.X, 4);                 % Find the number of images in the trai
for i = 1:train_img_no                                % loop over all train images to convert t
    train_32x32_grey.X(:,:,:,i) = rgb2gray(train_32x32.X(:,:,:,i));
end
```

Visualize the converted images dataset

```
figure;
test_slates_no = size(test_32x32.X, 4);     % find the number of image slates in the test datas
perm = randperm(test_slates_no,20);         % create an array of 20 random numbers
for i = 1:20                                % plot 20 random images from the dataset
    subplot(4,5,i);
    imshow(test_32x32_grey.X(:,:,:,perm(i)));
end
```



Reshape and normalize the images into a single vector

```
% Initialize the Test and Tran Image arrays in which to hold the reshaped normalised images
images_ts = nan(26032,1024);
```

```
images_tr = nan(73257,1024);

% Loop over all test images to convert them to reshape and normalize
for i = 1:test_img_no
    test_32x32_grey_reshaped = reshape(test_32x32_grey.X(:,:,:,i),[1024,1]); % reshape into an
    images_ts(i,:) = double(test_32x32_grey_reshaped)/255; % normalize to have the pixel values
end

% Loop over all train images to convert them to reshape and normalize
for i = 1:train_img_no
    train_32x32_grey_reshaped = reshape(train_32x32_grey.X(:,:,:,i),[1024,1]); % reshape into a
    images_tr(i,:) = double(train_32x32_grey_reshaped)/255; % normalize to have the pixel value
end
```

b) Preparing the labels for the RBM Model

Create arrays with OneHot encoding of the labels (ref)

Labels are structured in an one-hot array with 10 columns for each-digit e.g. the third column contains 1s (on) in the rows where the label for the letter 2 and 0s (off otherwise).

```
labels_ts = transpose(full(ind2vec(test_32x32.y',10)));     % Test labels with one-hot labels f
labels_tr = transpose(full(ind2vec(train_32x32.y',10)));    % Train labels with one-hot labels
Nd = size(images_ts,1)                                        % Number of training data points
```

```
Nd = 26032
```

```
Ni = size(images_ts,2)                                      % Size of the input vector
```

```
Ni = 1024
```

Training the Restricted Boltzman

(d) Initialize a Restricted Boltzmann Machine

The following sections build upon Andrea Valent's RBM implementation in MATLAB ref

```
% Add path to RBM implementation % forked from https://github.com/Andrea-V/Restricted-Boltzmann
addpath(genpath([pwd '\RBM'])) ;

% Initlialize the RBM arrays
[M, b, c] = rbm_init(Ni, 100) % Ni = size of the input vector, number of hidden units = 100
```

```
M = 1024×100
    0.0004   -0.0098   -0.0090    0.0042   -0.0177   -0.0083    0.0091    0.0015 ···
    0.0133    0.0034   -0.0059    0.0035   -0.0023    0.0024    0.0009   -0.0084
   -0.0276    0.0204    0.0187    0.0016   -0.0018   -0.0151    0.0028   -0.0134
    0.0036   -0.0182   -0.0097   -0.0125    0.0100   -0.0038   -0.0116   -0.0123
   -0.0018   -0.0037    0.0045   -0.0184   -0.0205   -0.0030   -0.0037   -0.0180
   -0.0181   -0.0194    0.0032    0.0015   -0.0085    0.0107   -0.0162   -0.0132
   -0.0093    0.0080    0.0109    0.0095   -0.0007    0.0004   -0.0026    0.0068
   -0.0016    0.0091    0.0003   -0.0179   -0.0060   -0.0047    0.0064   -0.0110
    0.0308   -0.0216   -0.0297   -0.0271    0.0153   -0.0184   -0.0160   -0.0056
    0.0227    0.0144   -0.0135    0.0094   -0.0087   -0.0197   -0.0071    0.0106
```

```
     ⋮
b = 1024×1
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     ⋮
c = 100×1
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     ⋮
```

M, b, c are generated by the RMB initalisation function:

- M (26032 x 100) corresponds to the interaction term between visible units (26032 inputs) and hidden units (100)
- b corresponds to the field term associated with visible units (initialised as zeros - corresponding to no activation)
- c corresponds to the field term associated with hidden units (initialised as zeros - corresponding to no activation)

(d) Training the Restricted Boltzmann Machine

To begin with we will use the following initial settings (10 minutes training time):

- cd_k = 1; (contrastive-divergence steps)
- eta = 0.01; (learning rate)
- alpha = 0.5; (momentum)
- lambda = 1e-5; (regularization)
- max_epochs = 5 (number of training epochs)

```
[M , b, c , errors] = rbm_train(images_ts, M, b, c, 1, 0.01, 0.5, 1e-5, 5)
```

```
-- shuffling inputs
-- training...
- epoch 0, error: 4.125456
-- shuffling inputs
-- training...
```

```
- epoch 1, error: 3.172930
-- shuffling inputs
-- training...
- epoch 2, error: 3.075481
-- shuffling inputs
-- training...
- epoch 3, error: 3.035261
-- shuffling inputs
-- training...
- epoch 4, error: 3.011611
-- shuffling inputs
-- training...
- epoch 5, error: 2.993863
-- shuffling inputs
-- training...
- epoch 6, error: 2.986387
M = 1024×100
    0.0232    0.0815   -0.0902    0.0019   -0.0052    0.1096    0.0537    0.0092 ⋯
    0.0142    0.0942   -0.0746   -0.0117   -0.0085    0.1324    0.0446    0.0185
    0.0103    0.0963   -0.0584   -0.0361   -0.0059    0.1606    0.0303    0.0248
    0.0060    0.0885   -0.0484   -0.0584   -0.0003    0.1821    0.0134    0.0220
   -0.0002    0.0771   -0.0499   -0.0759   -0.0036    0.1862    0.0006    0.0087
   -0.0051    0.0650   -0.0597   -0.0784   -0.0169    0.1701    0.0020    0.0009
   -0.0114    0.0541   -0.0760   -0.0758   -0.0298    0.1382    0.0109    0.0007
   -0.0165    0.0433   -0.0979   -0.0733   -0.0401    0.1003    0.0245    0.0057
   -0.0184    0.0342   -0.1242   -0.0659   -0.0388    0.0689    0.0375    0.0141
   -0.0158    0.0305   -0.1464   -0.0629   -0.0241    0.0436    0.0442    0.0216
      ⋮

b = 1024×1
    0.9648
    0.9528
    0.9350
    0.9241
    0.9152
    0.9125
    0.9012
    0.8981
    0.8957
    0.8922
      ⋮

c = 100×1
   17.9612
   12.3217
   17.8723
   18.8188
   15.9893
   13.2198
   18.6086
    8.5723
   12.5963
   19.2997
      ⋮

errors = 1×8
       Inf    4.1255    3.1729    3.0755    3.0353    3.0116    2.9939    2.9864
```

(e) Building the encoded representation

Using the trained Boltzmann model we will build an encoded representations of the training and test datasets

```
img_codes_tr = rbm_encode(images_tr, M, b, c) % encoded representation of the training dataset
```

```
img_codes_tr = 73257×100
    0.9999    0.0043    1.0000    1.0000    0.9997    0.9850    1.0000    0.0230 ···
    0.9973    0.0028    1.0000    1.0000    0.9982    0.8864    1.0000    0.0000
    0.0000    0.0000    0.6966    0.0093    0.0005    0.0000    0.0002    0.0000
    0.0000    0.0000    0.7393    0.0084    0.0017    0.0000    0.0172    0.0000
    0.4712    0.0000    0.9977    0.9527    0.0112    0.0038    0.7306    0.0000
    0.4717    0.0000    0.9997    0.9977    0.0458    0.0001    0.9871    0.0000
    0.9906    0.0010    0.9966    0.9989    0.9953    0.0042    0.9802    0.0000
    0.9994    0.0592    0.9996    0.9937    0.6395    0.0040    0.9990    0.2900
    0.9812    0.0000    1.0000    1.0000    0.9918    0.4153    0.9998    0.0002
    0.9991    0.0000    0.9995    0.9999    0.9679    0.2018    1.0000    0.9991
       ⋮
```

```
img_codes_ts = rbm_encode(images_ts, M, b, c) % encoded representation of the test dataset
```

```
img_codes_ts = 26032×100
    0.9977    0.1114    1.0000    1.0000    0.9974    0.8827    1.0000    0.0000 ···
    0.0081    0.0000    0.9999    0.3391    0.0001    0.0000    0.0578    0.0000
    0.2009    0.0000    0.8886    0.2437    0.0106    0.0000    0.0407    0.0000
    0.0462    0.0000    0.9999    0.7541    0.0009    0.0000    0.0508    0.0000
    0.0739    0.0000    0.8984    0.7718    0.0017    0.0000    0.2532    0.0000
    0.7842    0.0000    0.9998    0.9992    0.2612    0.0206    0.9917    0.0009
    1.0000    0.9978    1.0000    1.0000    1.0000    0.9992    1.0000    0.9888
    0.9928    0.2529    1.0000    0.9999    1.0000    0.0063    0.9996    0.0000
    0.7962    0.0000    0.9839    0.6689    0.0000    0.0000    0.6803    0.0000
    0.6119    0.0000    0.9952    0.5284    0.0893    0.0008    0.9843    0.0000
       ⋮
```

Learning the digits

(f) Initialize a one-layer artificial neural net (ANN)

We will use the encoded representation of the model to train a shallow model using patternnet

Pattern recognition networks are feedforward networks that can be trained to classify inputs according to target classes. The target data for pattern recognition networks should consist of vectors of all zero values except for a 1 in element `i`, where `i` is the class they are to represent (ref).

```
ANN = patternnet(100) % build a neural network with one hidden layer and one output layer.
```

```
ANN = 

    Neural Network

              name: 'Pattern Recognition Neural Network'
          userdata: (your custom info)

    dimensions:

         numInputs: 1
         numLayers: 2
        numOutputs: 1
    numInputDelays: 0
    numLayerDelays: 0
 numFeedbackDelays: 0
 numWeightElements: 100
        sampleTime: 1
```

```
connections:

   biasConnect: [1; 1]
  inputConnect: [1; 0]
  layerConnect: [0 0; 1 0]
 outputConnect: [0 1]

subobjects:

         input: Equivalent to inputs{1}
        output: Equivalent to outputs{2}

        inputs: {1x1 cell array of 1 input}
        layers: {2x1 cell array of 2 layers}
       outputs: {1x2 cell array of 1 output}
        biases: {2x1 cell array of 2 biases}
  inputWeights: {2x1 cell array of 1 weight}
  layerWeights: {2x2 cell array of 1 weight}

functions:

      adaptFcn: 'adaptwb'
    adaptParam: (none)
      derivFcn: 'defaultderiv'
     divideFcn: 'dividerand'
   divideParam: .trainRatio, .valRatio, .testRatio
    divideMode: 'sample'
       initFcn: 'initlay'
    performFcn: 'crossentropy'
  performParam: .regularization, .normalization
      plotFcns: {'plotperform', plottrainstate, ploterrhist,
                plotconfusion, plotroc}
    plotParams: {1x5 cell array of 5 params}
      trainFcn: 'trainscg'
    trainParam: .showWindow, .showCommandLine, .show, .epochs,
                .time, .goal, .min_grad, .max_fail, .sigma,
                .lambda

weight and bias values:

            IW: {2x1 cell} containing 1 input weight matrix
            LW: {2x2 cell} containing 1 layer weight matrix
             b: {2x1 cell} containing 2 bias vectors

methods:

         adapt: Learn while in continuous use
     configure: Configure inputs & outputs
        gensim: Generate Simulink model
          init: Initialize weights & biases
       perform: Calculate performance
           sim: Evaluate network outputs given inputs
         train: Train network with examples
          view: View diagram
   unconfigure: Unconfigure inputs & outputs
```

(f) Training the shallow ANN

We can now train the shallow neural net (softmax)

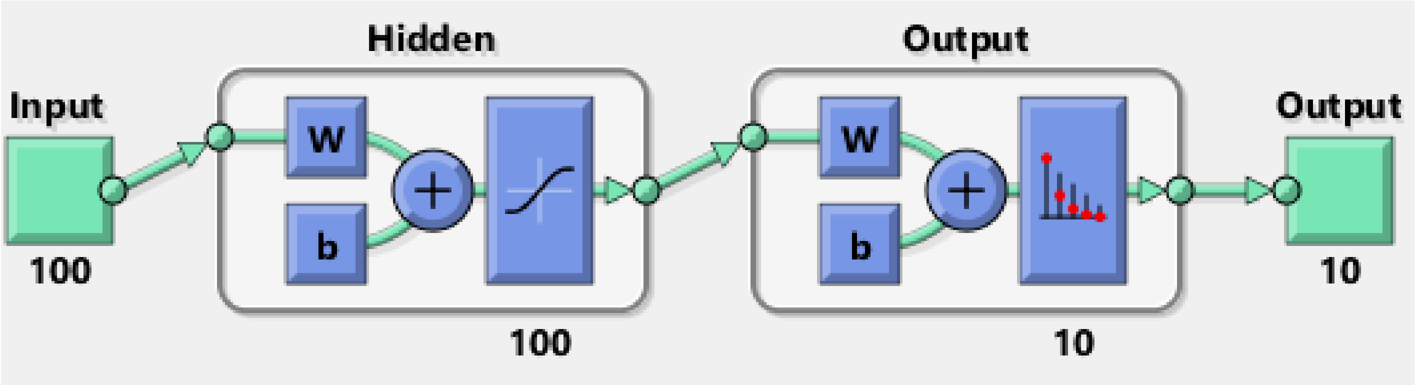
```
ANN_trained = train(ANN,img_codes_tr',labels_tr')
```

```
ANN_trained =

    Neural Network

              name: 'Pattern Recognition Neural Network'
          userdata: (your custom info)

    dimensions:

         numInputs: 1
         numLayers: 2
        numOutputs: 1
    numInputDelays: 0
    numLayerDelays: 0
 numFeedbackDelays: 0
 numWeightElements: 11110
        sampleTime: 1

    connections:

       biasConnect: [1; 1]
      inputConnect: [1; 0]
      layerConnect: [0 0; 1 0]
     outputConnect: [0 1]

    subobjects:

             input: Equivalent to inputs{1}
            output: Equivalent to outputs{2}

            inputs: {1x1 cell array of 1 input}
            layers: {2x1 cell array of 2 layers}
           outputs: {1x2 cell array of 1 output}
            biases: {2x1 cell array of 2 biases}
      inputWeights: {2x1 cell array of 1 weight}
      layerWeights: {2x2 cell array of 1 weight}

    functions:

          adaptFcn: 'adaptwb'
        adaptParam: (none)
          derivFcn: 'defaultderiv'
         divideFcn: 'dividerand'
       divideParam: .trainRatio, .valRatio, .testRatio
        divideMode: 'sample'
           initFcn: 'initlay'
        performFcn: 'crossentropy'
      performParam: .regularization, .normalization
          plotFcns: {'plotperform', plottrainstate, ploterrhist,
                    plotconfusion, plotroc}
        plotParams: {1x5 cell array of 5 params}
          trainFcn: 'trainscg'
        trainParam: .showWindow, .showCommandLine, .show, .epochs,
                    .time, .goal, .min_grad, .max_fail, .sigma,
                    .lambda

    weight and bias values:

                IW: {2x1 cell} containing 1 input weight matrix
                LW: {2x2 cell} containing 1 layer weight matrix
                 b: {2x1 cell} containing 2 bias vectors

    methods:
```

```
          adapt: Learn while in continuous use
      configure: Configure inputs & outputs
         gensim: Generate Simulink model
           init: Initialize weights & biases
        perform: Calculate performance
            sim: Evaluate network outputs given inputs
          train: Train network with examples
           view: View diagram
    unconfigure: Unconfigure inputs & outputs
```

```
view(ANN_trained);
```

The trained neural net has the following architecture



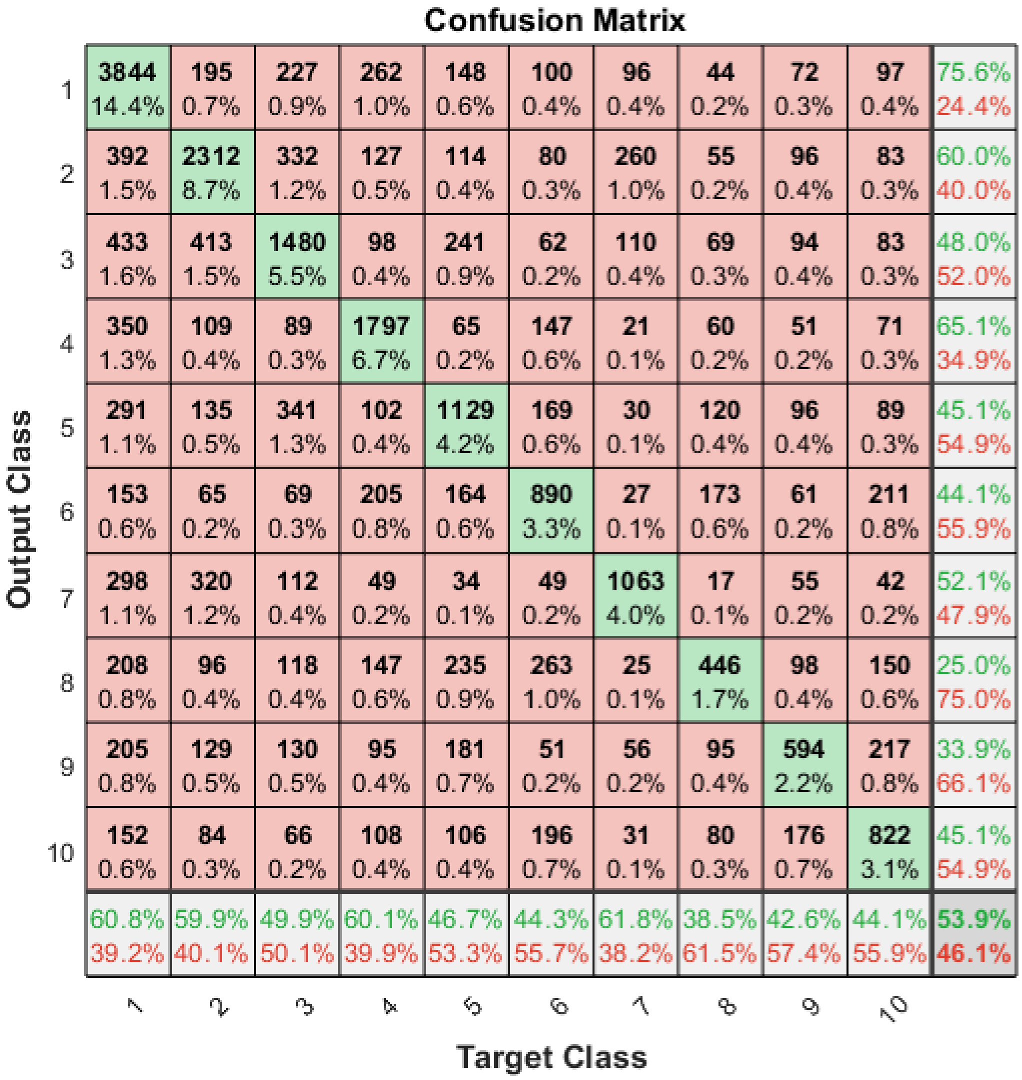
Testing the performance

(h) Confusion matrix for trained network on the training data

```
y_tr = ANN_trained(img_codes_tr') % Feeding the input training data through the trained neural
```

```
y = 10×73257
    0.5829    0.1185    0.1714    0.0004    0.0042    0.2162    0.3118    0.0168 ⋯
    0.0621    0.0571    0.0229    0.1560    0.8422    0.0041    0.2236    0.0377
    0.0521    0.0787    0.3929    0.1056    0.0606    0.0575    0.1197    0.2989
    0.0071    0.0317    0.0156    0.0585    0.0007    0.0478    0.1506    0.0118
    0.1103    0.1199    0.0316    0.0163    0.0013    0.4521    0.0714    0.3844
    0.0084    0.0435    0.0435    0.0510    0.0013    0.0694    0.0315    0.1037
    0.1445    0.0506    0.0011    0.0207    0.0816    0.0131    0.0240    0.0036
    0.0131    0.0703    0.3127    0.3182    0.0038    0.1012    0.0204    0.0363
    0.0163    0.1836    0.0075    0.2409    0.0013    0.0329    0.0343    0.0421
    0.0032    0.2462    0.0007    0.0324    0.0029    0.0057    0.0127    0.0648
```

```
plotconfusion(y_tr, labels_tr')  % Confusion matrix for trained network on the training data
```



	1	2	3	4	5	6	7	8	9	10	
1	3844 14.4%	195 0.7%	227 0.9%	262 1.0%	148 0.6%	100 0.4%	96 0.4%	44 0.2%	72 0.3%	97 0.4%	75.6% 24.4%
2	392 1.5%	2312 8.7%	332 1.2%	127 0.5%	114 0.4%	80 0.3%	260 1.0%	55 0.2%	96 0.4%	83 0.3%	60.0% 40.0%
3	433 1.6%	413 1.5%	1480 5.5%	98 0.4%	241 0.9%	62 0.2%	110 0.4%	69 0.3%	94 0.4%	83 0.3%	48.0% 52.0%
4	350 1.3%	109 0.4%	89 0.3%	1797 6.7%	65 0.2%	147 0.6%	21 0.1%	60 0.2%	51 0.2%	71 0.3%	65.1% 34.9%
5	291 1.1%	135 0.5%	341 1.3%	102 0.4%	1129 4.2%	169 0.6%	30 0.1%	120 0.4%	96 0.4%	89 0.3%	45.1% 54.9%
6	153 0.6%	65 0.2%	69 0.3%	205 0.8%	164 0.6%	890 3.3%	27 0.1%	173 0.6%	61 0.2%	211 0.8%	44.1% 55.9%
7	298 1.1%	320 1.2%	112 0.4%	49 0.2%	34 0.1%	49 0.2%	1063 4.0%	17 0.1%	55 0.2%	42 0.2%	52.1% 47.9%
8	208 0.8%	96 0.4%	118 0.4%	147 0.6%	235 0.9%	263 1.0%	25 0.1%	446 1.7%	98 0.4%	150 0.6%	25.0% 75.0%
9	205 0.8%	129 0.5%	130 0.5%	95 0.4%	181 0.7%	51 0.2%	56 0.2%	95 0.4%	594 2.2%	217 0.8%	33.9% 66.1%
10	152 0.6%	84 0.3%	66 0.2%	108 0.4%	106 0.4%	196 0.7%	31 0.1%	80 0.3%	176 0.7%	822 3.1%	45.1% 54.9%
	60.8% 39.2%	59.9% 40.1%	49.9% 50.1%	60.1% 39.9%	46.7% 53.3%	44.3% 55.7%	61.8% 38.2%	38.5% 61.5%	42.6% 57.4%	44.1% 55.9%	53.9% 46.1%

```
Warning: Targets were not all 1/0 values and have been rounded.
```

(i) Predicting the labels for the training dataset

Unlike in the case for y_training, we are now evaluating the RBM ML model using previously unseen test data

```
y_ts =  ANN_trained(img_codes_ts') % feeding the encoded test images data to the RBM ANN model
```

```
y_ts = 10×26032
    0.0775    0.0255    0.3020    0.1094    0.0158    0.8621    0.4538    0.9588 ···
    0.1847    0.5773    0.1721    0.0891    0.0496    0.0155    0.0400    0.0005
    0.0710    0.2278    0.0528    0.0482    0.0680    0.0030    0.0448    0.0004
    0.0296    0.0050    0.0003    0.1298    0.0385    0.0415    0.3151    0.0382
    0.0741    0.1188    0.0020    0.1744    0.1239    0.0046    0.0389    0.0004
    0.0755    0.0058    0.0001    0.0475    0.2541    0.0047    0.0108    0.0002
    0.0728    0.0293    0.4686    0.0630    0.0038    0.0385    0.0548    0.0000
    0.0573    0.0006    0.0019    0.0245    0.3710    0.0071    0.0117    0.0014
    0.1184    0.0029    0.0000    0.0359    0.0375    0.0156    0.0230    0.0002
    0.2391    0.0069    0.0001    0.2781    0.0379    0.0072    0.0070    0.0000
```

```
plotconfusion(y_ts, labels_ts')  % Confusion matrix for trained network on the training data
```

	1	2	3	4	5	6	7	8	9	10	
1	3547 13.6%	242 0.9%	266 1.0%	428 1.6%	121 0.5%	116 0.4%	133 0.5%	38 0.1%	113 0.4%	95 0.4%	69.6% 30.4%
2	548 2.1%	2142 8.2%	364 1.4%	236 0.9%	95 0.4%	97 0.4%	314 1.2%	88 0.3%	142 0.5%	123 0.5%	51.6% 48.4%
3	541 2.1%	340 1.3%	1100 4.2%	201 0.8%	187 0.7%	64 0.2%	125 0.5%	74 0.3%	166 0.6%	84 0.3%	38.2% 61.8%
4	238 0.9%	106 0.4%	83 0.3%	1684 6.5%	52 0.2%	157 0.6%	18 0.1%	43 0.2%	57 0.2%	85 0.3%	66.7% 33.3%
5	301 1.2%	112 0.4%	249 1.0%	201 0.8%	758 2.9%	242 0.9%	35 0.1%	171 0.7%	187 0.7%	128 0.5%	31.8% 68.2%
6	151 0.6%	93 0.4%	64 0.2%	332 1.3%	116 0.4%	829 3.2%	33 0.1%	122 0.5%	52 0.2%	185 0.7%	41.9% 58.1%
7	379 1.5%	349 1.3%	136 0.5%	57 0.2%	32 0.1%	48 0.2%	877 3.4%	19 0.1%	79 0.3%	43 0.2%	43.4% 56.6%
8	175 0.7%	97 0.4%	73 0.3%	244 0.9%	164 0.6%	311 1.2%	23 0.1%	289 1.1%	130 0.5%	154 0.6%	17.4% 82.6%
9	133 0.5%	91 0.3%	114 0.4%	180 0.7%	99 0.4%	78 0.3%	46 0.2%	94 0.4%	549 2.1%	211 0.8%	34.4% 65.6%
10	111 0.4%	104 0.4%	87 0.3%	177 0.7%	93 0.4%	176 0.7%	62 0.2%	72 0.3%	214 0.8%	648 2.5%	37.2% 62.8%
	57.9% 42.1%	58.3% 41.7%	43.4% 56.6%	45.0% 55.0%	44.1% 55.9%	39.1% 60.9%	52.6% 47.4%	28.6% 71.4%	32.5% 67.5%	36.9% 63.1%	47.7% 52.3%

```
Warning: Targets were not all 1/0 values and have been rounded.
```

Looking at the two confusion matrices above we can observe that:

- the erros are overall higher for the test dataset, which is to be expected as the model was bilnd to this set during testing
- errors do not increase significantly between the two matrices, suggesting that the model is generalizable and not overfitted to the test data
- the number that were confused the most was 0, especially with lower value digits like 0,1,2,3,4 -- this might also be due to the fact that the dataset is bias to the digit 0 (more of these examples than for other digits)

Visualizing the filters

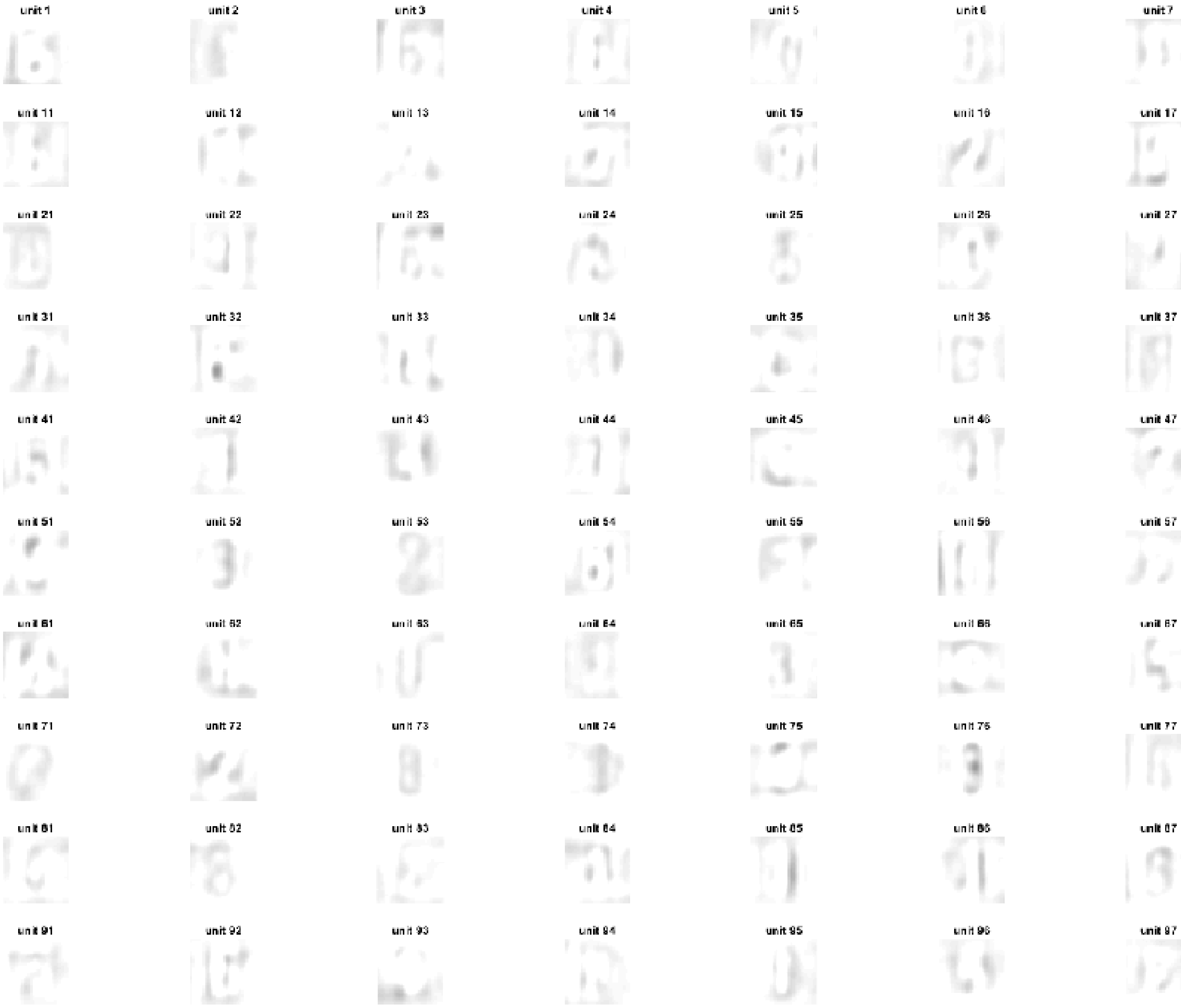
(j) Converting from M vector space back to the image space

To visualise the filters we will convert back from encoded space `M` to the original image space of 32 x 32 pixels

`M_images`

```
figure
hold on
for i=1:size(M,2)                          % Loop over all of the columns of the M matrix (all 100
   subplot(10, 10, i);                     % Create a subplot of the 100 filters into a 10×10 grid
   M_images = reshape(M(:,i), 32, 32);     % Reshape the filters in M by converting each of the fi
   imshow(1-M_images)                      % Swap the black and white colors to make the filters e
%    title(sprintf('unit %d', i));         % Used to generate the larger image included below
end
```

(k) Larger view of the 100 filters visualisation showing the pixel activation of each filter (black pixels)



(l) Identifying the most important filters

We can find the most important filters by looking at the coefficient vector, `c` which corresponds to the field term associated with hidden units.

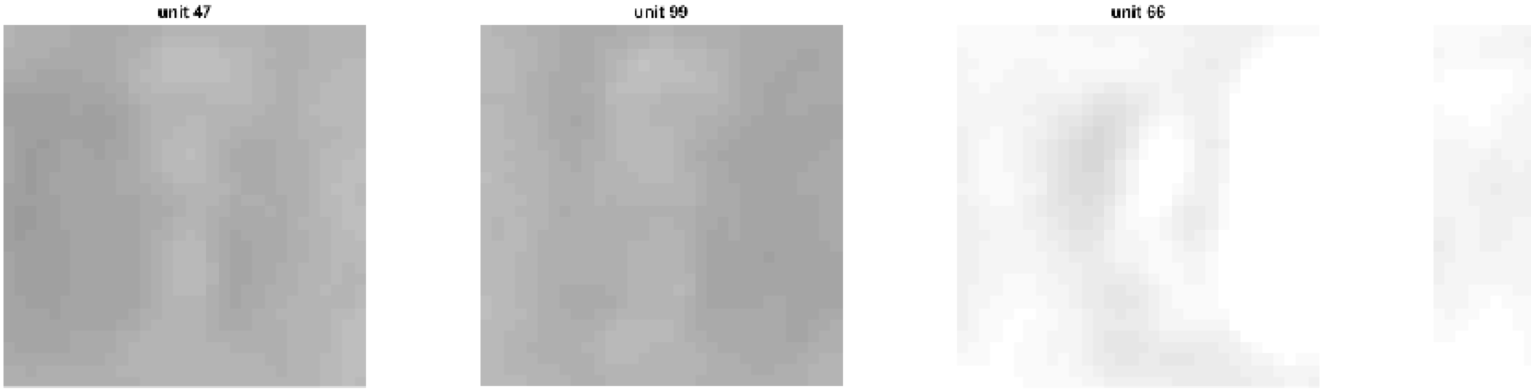
```
ntop = 5;                       % number of top filters to extract
[val, ind] = sort(c,'ascend');  % sort the filters
```

```
figure
hold on
for i=1:ntop                                      % Loop over all max activation ntop filters
```

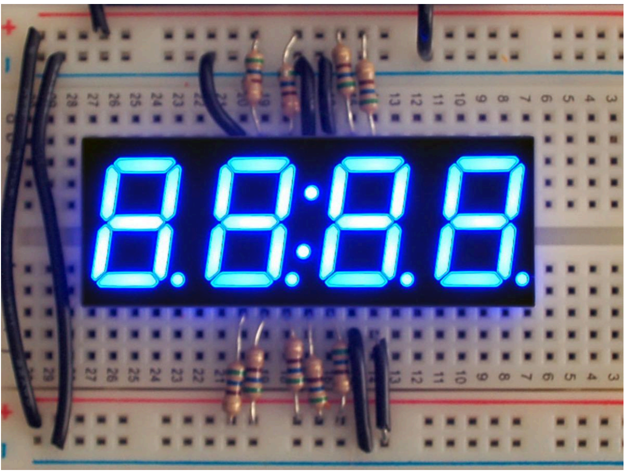
```
    subplot(1, 5, i);                               % Create a subplot of the max ntop filters int
    % Reshape the filters in M by converting each of the filter vector into an image and select
    M_image = reshape(M(:,ind(i)), 32, 32);
    imshow(1-M_image)                               % Swap the black and white colors to make the
    title(sprintf('unit %d', ind(i)));              % Include the index name
end
```

I think these are emerging as the top 5 most important filters because they are the most general, unlike other more specialised hidden layers, these activate for most digits in the dataset (ref). It is particularly interesting to observe `unit 57`, which has the shape used by many digital clocks displays (the 7-Segment) as it best encodes all digits using the minium number of lines.

The 7-Segment, similar to unit 57 in the Restricted Boltzmann Machine trained on the Street View House Numbers (SVHN) Dataset



5 RBM filters



7 segment clock display fully activated (ref)