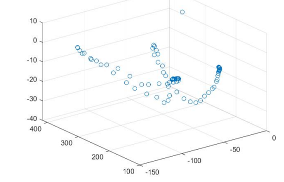
Digits 3D

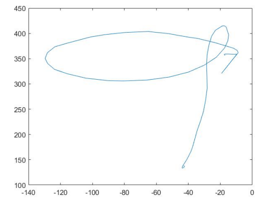
Group 8
Mark Ibrahim (0556829)
Aakif Nawaz (0555451)
Mickaël Denni (0556557)

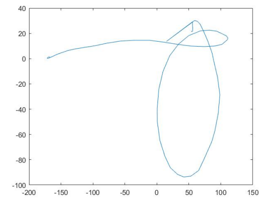
9th December 2019

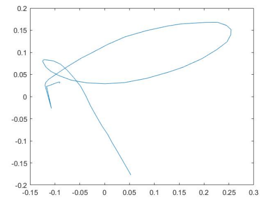
1 Preprocessing

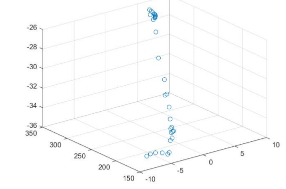
Feature extraction is the technique of dimension reduction of the given raw data to a more meaningful and useful information. For the current task, the 3D location information for each fingertip drawing a digit was measured by means of a LeapMotion Sensor. This information was provided to us in a set of 1000 *.mat* and *.csv* files comprising of the fingertip coordinates of each digit. We only used *.mat* files in our project because it was more suitable for us. Every file name provides information about the number of the sample that was selected and the associated digit.

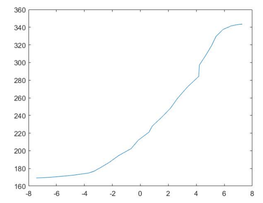

(a) Original 3D cloud plot of a **9**

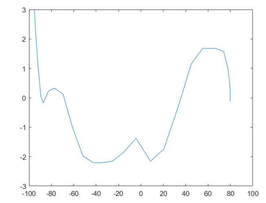

(b) X/Y plot of the same **9**


(c) plot after PCA of the same **9**


(d) plot after LLE of the same **9**


(e) Original 3D cloud plot of a **1**


(f) X/Y plot of the same **1**


(g) plot after PCA of the same **1**

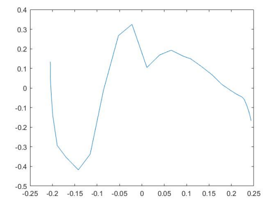

(h) plot after LLE of the same **1**

Figure 1: 2D reconstruction methods comparison for a **9** and a **1**

At the beginning, we wanted to reduce dimension (from 3D to 2D) of our samples. For that, we tried to implement some well known dimensionality reduction methods like **Principal Component Analysis** (**PCA**) and **Locally Linear Embedding** (**LLE**) to get a good representation of the 3D points. In Figure 1, we plotted 2D representations for a **9** and a **1** using three ways: X and Y coordinates, plots after PCA and plots after LLE. The PCA and LLE implementation described here were carried out with the aid of dimensionality reduction toolbox by L.J.P. van der Maaten[1]. However, as we can see, reducing dimension with advanced methods gives us poor result compared to plotting the X and Y coordinates. In fact, the relevant information of the provided samples vary along X and Y axes and not along the principal components of the data. We obtained fairly good 2D plots along the X and Y coordinates.

The plot obtained on its own is not enough to suit the purpose of our classifier. Several preprocessing steps are needed :

- Firstly, the plots obtained from the samples are unbounded i.e. they are spread along different coordinate axis limits. This is easily solved by normalizing the plot between 0 and 1.
- Secondly, each sample does not have the same number of points so we cannot compare them. That is why we decided to tranform our 2D graphs into images.
- Also, the line plots close to the axes in the normalized plot tend to merge with the axes lines. Additionally, the plot ticks which are visible along the x and y axes are a feature of Matlab plots and should not be a part of the image. It could potentially disrupt our classification algorithm and hence needs to be eliminated. To counter this issues we start off by increasing the axes limits followed by saving the graph as an image. By doing this, we crop the image borders to get rid of the ticks along the axes and save each of the samples as an image.
- Once we have our images saved, we need to reshape them to compress our data. As a result of that, we reshape our 343x434 images into 16x16 images.
- Now, we have low dimension images. We additionally convert the image into a binary image consisting only of black and white pixels by means of thresholding. In fact, classifiers work better with binary matrices and the level of gray of the pixels is not necessary to consider.
- In order to have a more useful way of accessing the database of images, we shrink our 16x16 matrices in a 256 vector so our data are stocked in a 256x1000 matrix.
- Then, in order to use classifier efficiently, we perform PCA along the data variables as a large number of pixels are insignificant to characterize a digit. Our data matrix size becomes 132x1000.
- Finally, we randomly split the data matrix into the training and testing sets with a ratio of 80%-20%.

Here is the Matlab Code implemented to preprocess data :

```matlab
%% Saving the plots as images

close all;
clear all;
```

```
files = dir('*.mat'); %collect all the .mat files

for k=1:1000
    filename = files(k).name;
    load(filename);

    %normalizing the data
    pos(:,1)=(pos(:,1)-min(pos(:,1)))/(max(pos(:,1))-min(pos(:,1)));
    pos(:,2)=(pos(:,2)-min(pos(:,2)))/(max(pos(:,2))-min(pos(:,2)));

    %we keep only the first two dimensions
    plot(pos(:,1),pos(:,2));

    %we enlarge a bit the graph so the plot does not match with the
        axes
    %in order to remove them later
    axis([-0.1 1.1 -0.1 1.1]);

    %we save the graph as an image
    F = getframe;
    A = F.cdata(:,:,1);
    [n,m]=size(A);

    % Obtaining an image with only two colors (black 1 and white 255)
    for i = 1:n
        for j = 1:m
            a = A(i,j);
            if a<200
                A(i,j)=1;
            else
                A(i,j)=255;
            end
        end
    end

    % Removing the border of the graph
    for i = 1:n
        for j= 1:m
            if i>10 & j>10 & (n-i)>10 & (m-j)>10
                B(i-10,j-10)=A(i,j);
            end
        end
    end

    filename = strcat(erase(filename,'mat'),'jpg'); %Restructure the
        file name
    imwrite(B,filename); %save the image in the current folder
end

%% Saving the data in a 2-dimensionnal matrix

close all;
clear  all;
files = dir('*.jpg'); %collect all .jpg images that we just saved

mat1616 = zeros(256,1000); %each line of mat1616 corresponds to one
    image
% of a sample that has been resize in a 16x16 image and stock in a
    single
% vector

for i = 1:1000
    img =  imread(files(i).name); %we read the image
    res = imresize(img,[16 16]); %we resize it into an 16x16 image
    mat1616(:,i) = double(res(:)); %we store it in mat1616 as one
        vector of length 256
end

save('mat1616.mat','mat1616'); %we save the matrix to use it for
    classification
```

2 Classification with Multilayer Perceptron

MultiLayer Perceptron (**MLP**) is a common class of Artificial Neural Networks (ANNs). In such a classifier, we have number of neurons in each layer that are connected to the neurons of the preceding and following layers. In addition, there is no connection between the neurons of the same layer. As a default construction for MLP you should have three layers : input layer, hidden layer and output layer. The hidden layer and the output layer are those that involve processing. This construction may differ from one application to another, as you may increase the number of hidden layers depending on the application and how many hidden layers is needed to process. However, adding additional hidden layers often leads to overfitting, that is why we decided to use only one hidden layer and instead vary the number of neurons.

In MLP, we apply feed-forward operations to calculate the weighted matrices between layers. Then the sum is fed through a nonlinear activation function for example by using different functions like the *sigmoid*, *tanh* or *ReLU* that are the most common ones. Using *ReLU* leads to a high divergence in Squared Sum Estimation (SSE) that can be explained by the fact that we should range our neurons value between -1 to 1 and not only from 0 to 1 like *ReLU* does. Even while using the *sigmoid* we could not get the highest accuracy. For example, when we used 200 neurons and 10000 epochs we got 91.5% of accuracy, on the other hand *tanh* gave us 95.5% of accuracy. That is why in our project we have used only *tanh* as the activation function, which maps the resulting values even if there is a negative value as the range of this function is from -1 to 1.

Also, we fixed the value of learning rate ρ to 0.0001. ρ will affect how fast the weighted matrices are modified for each epoch. So, we decided to fix a low value of learning parameter and instead adjust the number of epochs.

To determine the most suitable number of neurons, we fixed the number of epochs to 10 000 then we tried different values and listed the accuracies and execution times we get in Table 1. Although the accuracy was the main variable we wanted to maximize, the execution time of our classifier was also taken into consideration. The highest accuracy was obtained for a 200-neuron MLP : we achieved 95.50% of accuracy. Setting up the number of neurons more than 300, we observed overfitting phenomenon : even if the SSE seemed to decrease in the convergence graph, we obtained lower accuracy (94.50% of accuracy for a 300-neuron MLP).

Then, we wanted to determine the most appropriate number of epochs. In the same way, we set the number of neurons to 200 and varied the number of epochs and listed the accuracies and the execution times in Table 2. We acheived the best result for 8000 epochs : **an accuracy of 95.5% in 70sec**. We also observed overfitting phenomenon for number of epochs higher than 15 000.

The convergence graph and the confusion matrix that we obtain after classifying the data with a 200-neuron MLP and 8000 epochs are respectively in Figure 2 and Figure 3. In this confusion matrix, we can see for which classes the errors occur. For instance, class 1 is one of the classes that produce quite a lot of errors : 3 samples are missclassified in class 1. It surely comes from the fact that ones are the numbers for which the 2D reconstruction was relatively imprecise as we can see in (f) in Figure 1.

Here is the Matlab code implemented to run a multilayer perceptron :

```
close all;
clear all;
rng('default');

load('mat1616.mat');
% X is the data matrix
X = mat1616;
clearvars mat1616

% Y corresponds to the desired outputs
Y = [zeros(1,100) ones(1,100) repmat(2,1,100) repmat(3,1,100) repmat
    (4,1,100)...
     repmat(5,1,100) repmat(6,1,100) repmat(7,1,100) repmat(8,1,100)
         repmat(9,1,100)];

%% Plot 6 random samples

% % figure(1)
% % for ii = 1:6
% %      subplot(2,3,ii)
% %      rand_num = randperm(1000,1); %select one random number
% %      imshow(uint8(reshape(X(:,rand_num),16,16))) %we reshape the 256
      vector in 16x16 matrix to show it
% %      title((Y(rand_num)),'FontSize',20)
% %      axis off
% % end
% % colormap gray

%% Transforming a bit data matrix X

% All the white value have now the value 1 and black pixels value 0
% Classifiers work better with binary matrix
for i=1:size(X,1)
    for j=1:size(X,2)
        a = X(i,j);
        if a ~= 255
            X(i,j) = 0;
        else
            X(i,j) = 1;
        end
    end
end

clearvars a

```

```
%% Principal component analysis to reduce dimensions of the samples

%some pixels are useless to characterize corresponding numbers
%that is why we reduce dimension so we deal with less data
[coeff,score,latent] = pca(X');
latent = latent/sum(latent);
index = find(cumsum(latent)>0.95); % we keep dimensions that represent
    95% of the data
index(1) %we keep 132 dimensions out of 256
X=score(:,1:index(1))';

%% Dividing the training and testing sets

n = 1000; %number of samples
P = 0.80 ; % 80% of the data for training
idx = randperm(n); % random division

Xtrain = X(:,idx(1:round(P*n)));
Ytrain = Y(idx(1:round(P*n)));
Xtest = X(:,idx(round(P*n)+1:end));
Ytest = Y(idx(round(P*n)+1:end));

clearvars n P idx

%% MultiLayer Perceptron

%class 0 becomes class 10 in order to deal with output matrix
Ytrain(Ytrain == 0) = 10;
Ytest(Ytest == 0) = 10;

tic
[ypred, t, wHidden, wOutput] = mlp(Xtrain, Ytrain, Xtest, 10000);
execution_time = toc %execution time of the classifier

accuracy = sum(Ytest == ypred)/length(Ytest)

%Confusion matrix
C = confusionmat(Ytest,ypred);
confusionchart(C,[1:9 0]);
```

3 Observations

Number of Neurons	Accuracy	Execution time
5	86.00%	5sec
10	92.00%	6sec
50	94.50%	20sec
100	94.50%	37sec
150	95.00%	46sec
175	95.00%	82sec
200	95.50%	92sec
225	95.50%	100sec
250	95.50%	108sec
300	94.50%	134sec

Table 1: Accuracies and Execution time of MPL classifiers (10 000 epochs) depending on the number of neurons

Number of Epochs	Accuracy	Execution time
1000	90.50%	9sec
5000	94.00%	43sec
6000	95.00%	61sec
7000	95.00%	53sec
8000	95.50%	70sec
10 000	95.50%	92sec
15 000	94.50%	136sec
20 000	95.00%	181sec

Table 2: Accuracies and Execution time of MPL classifiers (200 neurons) depending on the number of epochs

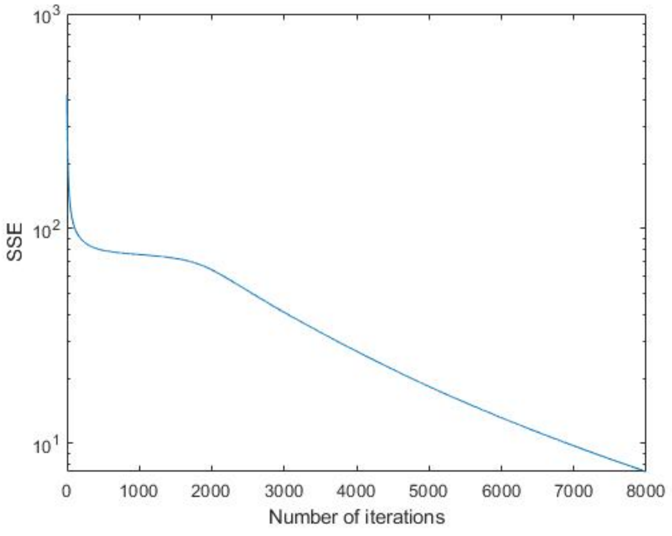
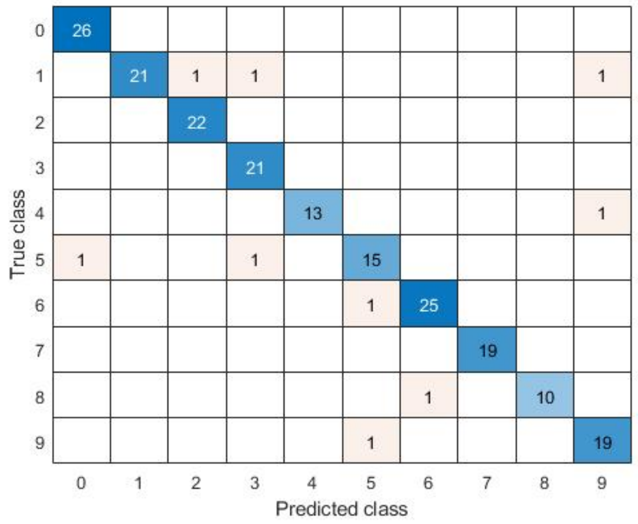


Figure 2: Convergence of 200-neuron MLP classifier with 8 000 epochs



True class
Predicted class

True class \ Predicted class	0	1	2	3	4	5	6	7	8	9
0	26									
1		21	1	1						1
2			22							
3				21						
4					13					1
5	1			1		15				
6						1	25			
7								19		
8							1		10	
9						1				19

Figure 3: Confusion matrix obtained after 200-neuron MLP classification with 8 000 epochs

References

[1] E.O. Postma L.J.P. van der Maaten and H.J. van den Herik. Dimensionality reduction: A comparative review. *Tilburg University Technical Report, TiCC-TR 2009-005*, 2009.