ECE 739: Computer Experiment Report

Neural Networks & Learning Machines:

Examples of Pattern Classification using Neural Networks trained by the Extended Kalman Filter and Support Vector Machines

Tuesday April 26, 2011

Dr. Simon Haykin

Jonathan Hernandez
0646927

Robert Tisma
0658942

Contents

Chapter 1

Introduction

In the field of signal processing, obtaining general statistical information on data sets is only useful for simple tasks such as determining mean, variance, median etc. However, for tasks involving pattern recognition or system modelling, simple statistics provide insufficient information. In general, pattern recognition is where objects are classified into a number of categories or classes using a mapping function trained by labelled data [1]. On the other hand, system modeling or function approximation is where labelled training data is used to approximate an unknown input-output mapping function, such as in regression. [2]. In both variants, the important goal is to generate mapping functions that very accurately mimic the true mapping functions with as little error as possible.

1.1 Support Vector Machines

In this report, pattern classification is considered for two different learning machines. One learning machine is the Support Vector Machine (SVM). Essentially the SVM is a classifier that uses a $(p-1)$-dimensional hyperplane to separate a p-dimensional feature vector into different classes [3]. The SVM determines the optimal $(p-1)$-hyperplane by finding its optimal position and orientation that maximizes the distance to the nearest data point of each class (also known as *support vectors*) [6]. An example of a binary SVM is shown in Figure 1.1, where one class is assigned the value of $y_i = +1$ and the other class $y_i = -1$.

If the labelled training data set is defined as in Equation (1.1), then the hyperplane is defined using Equations (1.2), where x and y represent the input data and the class label, respectively. Equations (1.3) and (1.4) represent the classification of the positive class, $y_i = +1$, and the negative class, $y_i = -1$, respectively, where $\mathbf{x_i}$ represents the training data. Equation (1.5) represents the combined form of Equations (1.3) and (1.4).

$$D = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_m, y_m)\}, x \epsilon \mathbb{R}^p, y \epsilon \{-1, 1\} \tag{1.1}$$

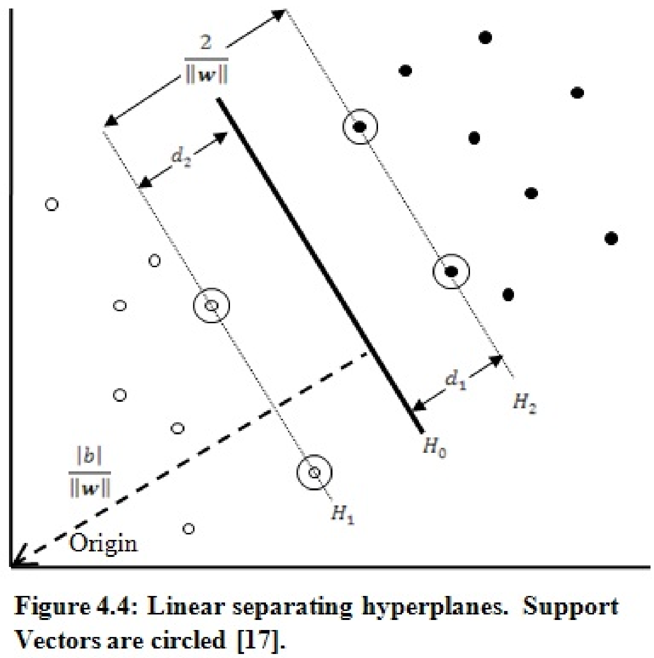

Figure 1.1: Example of a Support Vector Machine [5].

$$\langle \mathbf{w},\mathbf{x} \rangle + b = 0 \tag{1.2}$$
$$\langle \mathbf{w},\mathbf{x_i} \rangle + b \geq +1 \quad y_i = +1 \tag{1.3}$$
$$\langle \mathbf{w},\mathbf{x_i} \rangle + b \leq -1 \quad y_i = -1 \tag{1.4}$$
$$y_i(\langle \mathbf{w},\mathbf{x_i} \rangle + b) - 1 \leq 0 \tag{1.5}$$

Since the width of the margin is defined by $\frac{2}{\|\mathbf{w}\|}$, the SVM calculates the optimal hyperplane by finding the parameters $\mathbf{w}$ and b that maximize the margin. The optimization is done by minimizing the Lagrangian function. At the conclusion of the optimization, a specific set of weights called, Lagrangian multipliers are obtained. These are then used for the binary classifier defined in Equation (1.6). A detailed explanation of these mathematical concepts is described in [5] and [6].

$$f(\mathbf{x}) = sgn\left(\sum_{i=1}^{m} \alpha_i y_i \langle \mathbf{x}, \mathbf{x_i} \rangle + b\right) \tag{1.6}$$

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases} \tag{1.7}$$

1.2 Multilayer Perceptron using the Extended Kalman Filter

The other learning machine is the Multilayer Perceptron (MLP) which is trained using the Extended Kalman Filter (EKF). The MLP architecture is composed of 3 essential layers: the input layer, the hidden layer(s) and the output layer (Figure 1.2).

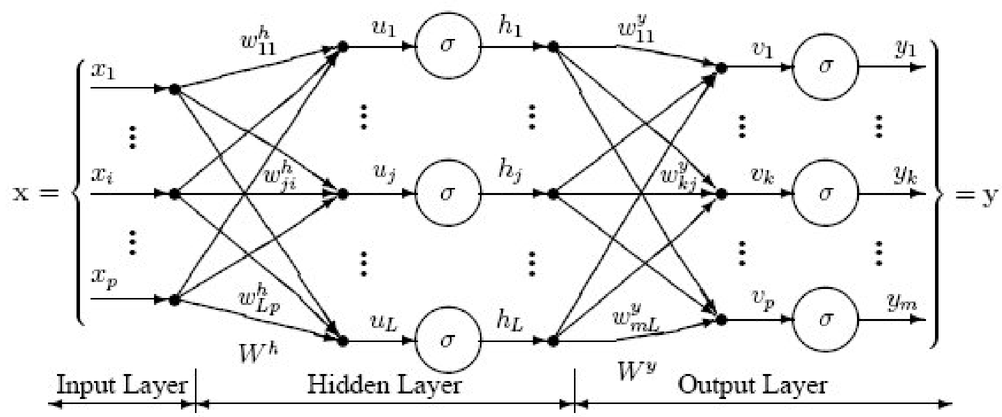

Figure 1.2: Example of a Multilayer Perceptron [4].

Each neuron is composed of a set of synaptic weights, $\mathbf{w_l}$ which are multiplied by the output of the neurons in the previous layer as shown in Figure 1.2. The general equation governing the calculation is shown in Equation (1.8), where L is the total number of layers.

$$F(\mathbf{x}, \mathbf{w}) = \varphi(\mathbf{w}^T_{\mathbf{(L)}}\varphi(\mathbf{w}^T_{\mathbf{(L\text{-}1)}}\varphi(\cdots\varphi(\mathbf{w}^T_{\mathbf{(1)}}\mathbf{x})))) \tag{1.8}$$

The activation function, $\varphi(\cdot)$, is a nonlinear function that limits the amplitude of the output of a neuron. A common activation function is the hyperbolic tangent function as shown in Equation (1.9), where a and b are adjustable parameters.

$$\varphi(u) = a\tanh(bu) \tag{1.9}$$

In order to minimize the error in classifying data, the optimal synaptic weights have to be adjusted using a learning algorithm. In this experiment, the EKF was used to obtain the optimal synaptic weight that reduces the total error associated with the MLP network. The EKF is the nonlinear version of the Kalman Filter, which utilizes the same principles by using measurements containing noise and calculating approximations to the true values associated with the measurement. An in depth discussion and mathematical formulation regarding the EKF can be found in [7].

Chapter 2

Problem & Methodology

Consider the classification problem in Figure 2.1. The task is to compare the performance between the SVM method and the MLP-EKF method, by classifying data into 2 classes: the red region is labelled $y_i = +1$ and the blue region is labelled $y_i = -1$. The radii are defined as: $\mathbf{r} = [r_1, r_2, r_3] = [0.2, 0.5, 0.8]$. For the SVM implementation, the goal is to train the system using 100 epochs, where each epoch consists of 200 randomly distributed training examples. Using the constraint, $0 \leq \alpha_i \leq C$, the SVM must be trained using the values, $C \epsilon \{100, 500, 2500\}$ for all i. In addition, the classification error must be determined as well as the decision boundary. Similarly, the MLP-EKF method requires the same training as well as construction of the decision boundary and calculation of the classification error.

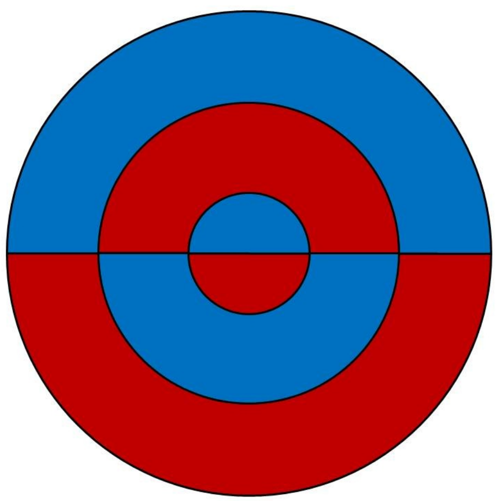

Figure 2.1: The classification problem [2].

2.1 Procedure

The first step required is to generate both training and testing data, as shown in A.2. This was done by generating random radii and angles for each data point, which was then labelled to the appropriate class. The data was then converted from polar coordinated to cartesian coordinates to properly simulate the desired problem shown in Figure 2.1. Figure 2.2 shows the generated data. Once the data was generated, a SVM model was created using the function, *SVM* shown in

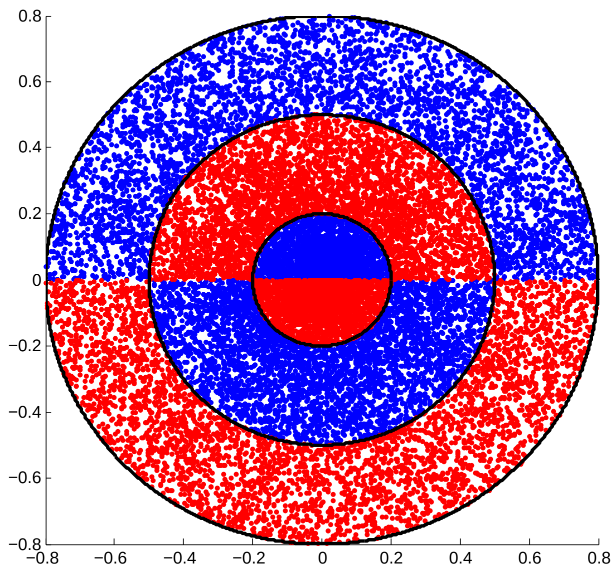

Figure 2.2: 10000 samples of generated data.

A.3. With in the function, the training data was used to train the SVM structure using the RBF kernel, and then the testing data was used to evaluate the performance of the SVM classification. In addition, a plot of the classification of the test data is created along with the percentage of errors associated with each epoch.

For the classification using the MLP-EKF method, the function *MLPEKF* shown in A.6 was used. This function takes the state vector, θ (which contains all the information regarding synaptic weights and biases), training data, $\mathbf{x}$, state covariance matrix, $\mathbf{P}$, measurement covariance matrix, $\mathbf{R}$, process covariance matrix, $\mathbf{Q}$, and the classification data of $\mathbf{x}$, $\mathbf{y}$, to train the MLP using the EKF. At first, θ and $\mathbf{x}$ are used to calculate the initial output of the MLP. The output is then used as an input along with θ, $\mathbf{P}$, $\mathbf{Q}$ and $\mathbf{R}$ for the EKF function, *ekf*, described in A.7. The EKF then calculates a predicted estimate of θ which approximates the ideal weight and bias values of the network for the current previous inputs. In the next step, the output of *MLPEKF* is then used to construct the network model. The testing data is then passed though the neural network, resulting in an output containing classification data. The classification data as a result of the model is then compared with the true test data classification. All classification mismatches are recorded as errors, resulting in an error performance vector, which contains the percentage errors for each epoch. All the above functions are called in the main m-file called *MAIN*, as shown in A.1.

Chapter 3

Results

3.1 SVM Performance

The SVM implementation proved to be extremely successful. The training consisted of 100 epochs of 200 data points using different values for C. Figure 3.1 demonstrates the classification of the test data with $C = 100$ and an error rate of 1.0%.

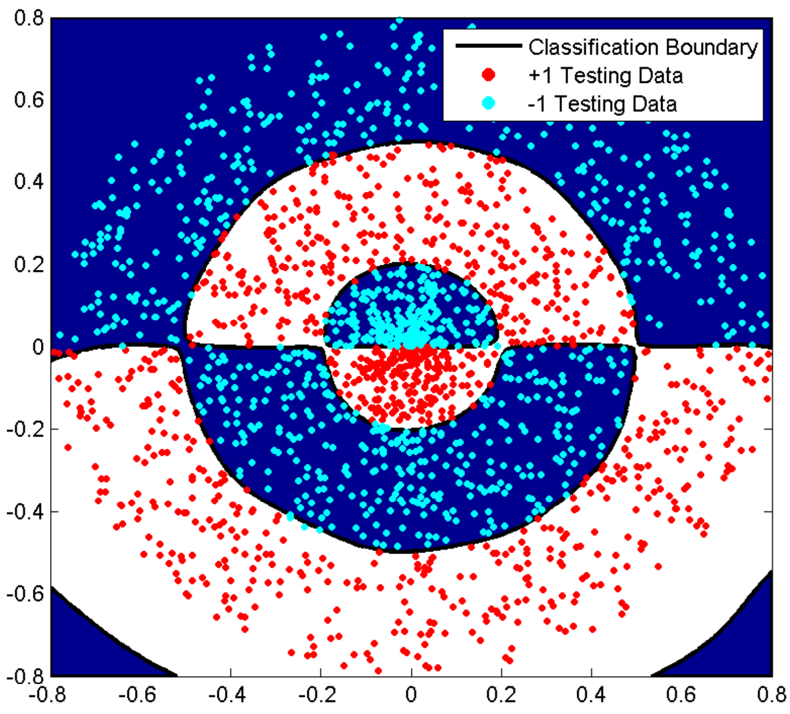

Figure 3.1: Result of classification on 200 samples of test data with $C = 100$.

Figure 3.2 demonstrates the classification of the test data with $C = 500$ and an error rate of 1.4%.

Figure 3.3 demonstrates the classification of the test data with $C = 2500$ and an error rate of 1.5%.

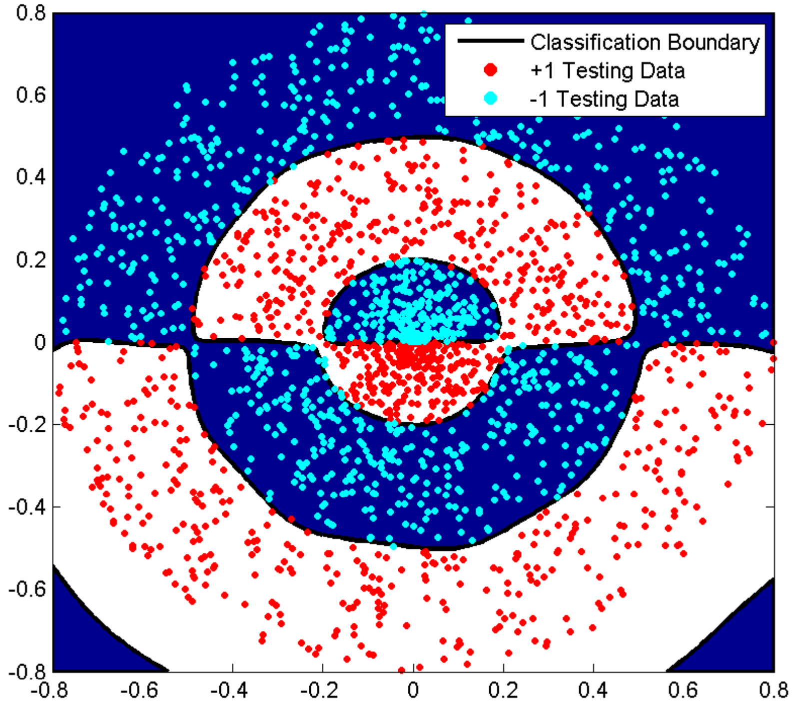

Figure 3.2: Result of classification on 200 samples of test data with $C = 500$.

Figures 3.1 to 3.3 represent an SVM trained with 10 epochs, where the blue regions represent class(-1) and the white regions represent class(+1). The coloured data points demonstrate the true classification of the testing data, where red represents the true classifications of class(+1) and blue represents the true classifications of class(-1). Therefore, any red points located in the blue region or any blue points located in the white region, constitutes a classification error by the SVM. As the number of epochs of training approaches 100, the figures begin to look identical as the errors are much less than 1%. After the SVM was trained with 100 epochs of data (200 samples each), the classification results on the test data demonstrated that the greater C is, the lower the rate of error, as shown in Figure 3.4.

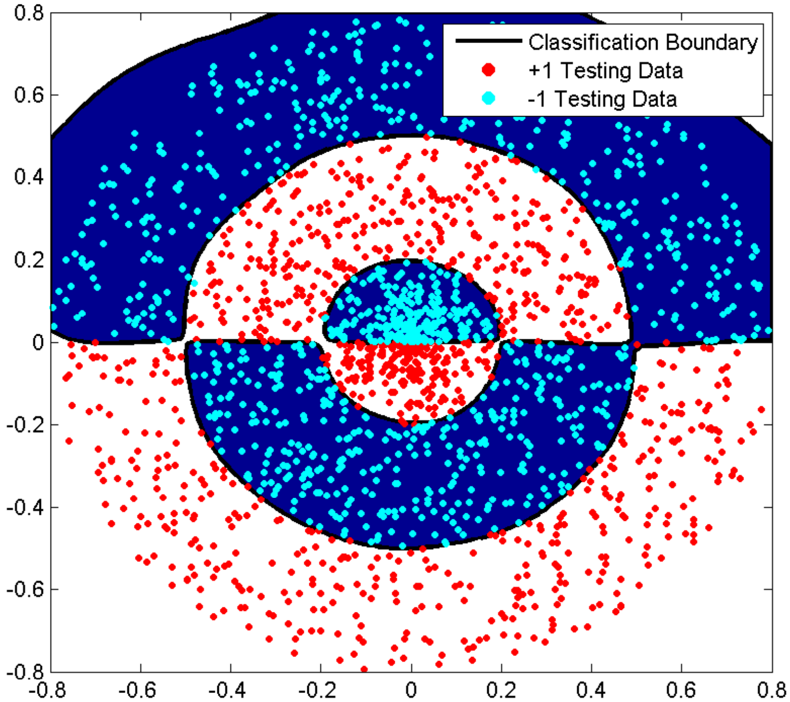

Figure 3.3: Result of classification on 200 samples of test data with $C = 2500$.

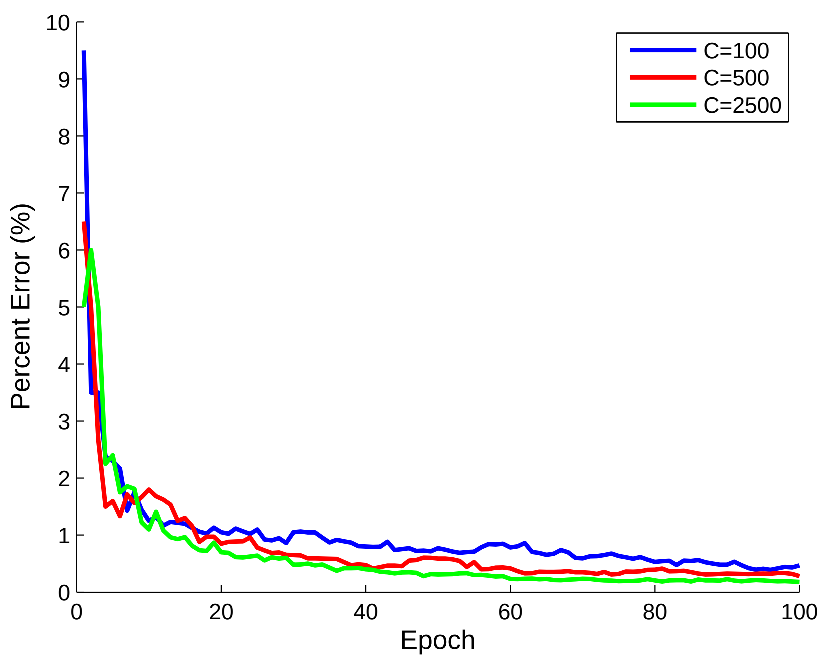

Figure 3.4: Learning rate curve of the SVM for different values of C.

3.2 MLP-EKF Performance

For the MLP-EKF implementation, the results showed poor performance using the same amount of training data as in the SVM implementation. For the testing phase, the number of samples was increased to 20000 to show the classification boundaries. Figure 3.5 demonstrates the classification of the test data with an error rate of 40.92%.

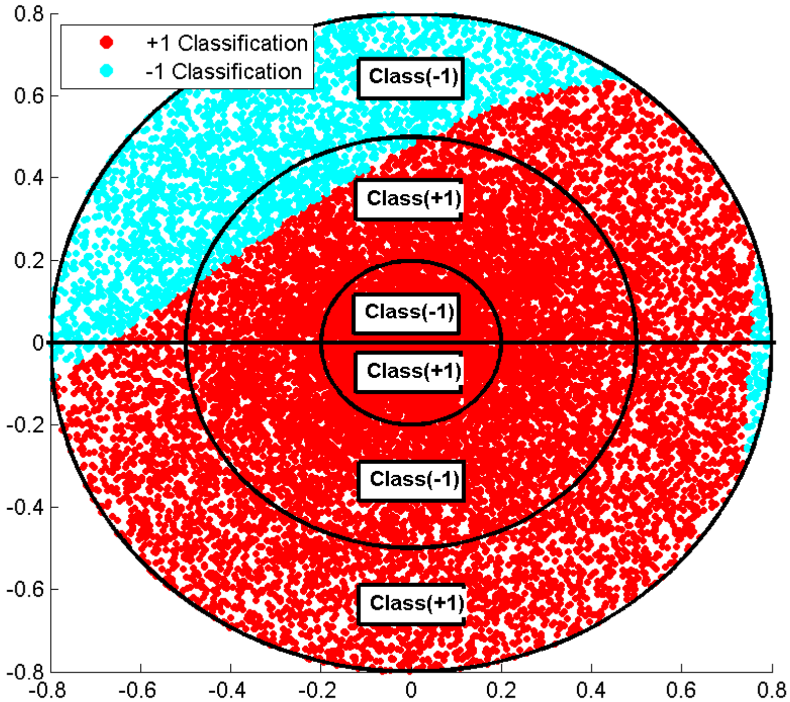

Figure 3.5: Result of classification on 20000 samples of test data using an MLP trained with 100 epochs.

Figure 3.6 demonstrates the classification of the test data using an SVM trained with 4000 epochs instead of the common 100. Since the error rate in Figure 3.5 was very high, the number of epochs used for training was increased in an attempt to provide a lower error rate, which was 18.34%.

In Figures 3.5 and 3.6, the regions bounded by black lines provide the true classification for data points, as shown by the labels. The red data points represent test data that was classified as belonging to class(+1) and blue data points represent test data that was classified as belonging to class(-1). Therefore, any red data points located in the regions labelled "Class(-1)" or any blue data points located in the regions labelled "Class(+1)" constitutes a classification error by the MLP. Figure 3.7 plots the learning rate curve of the MLP for 4000 epochs with 200 data point in each. It is evident that the rate of learning for the MLP is extremely slow and converges to about a 21% error rate, which is still much greater than that for the SVM trained on only 100 epochs.

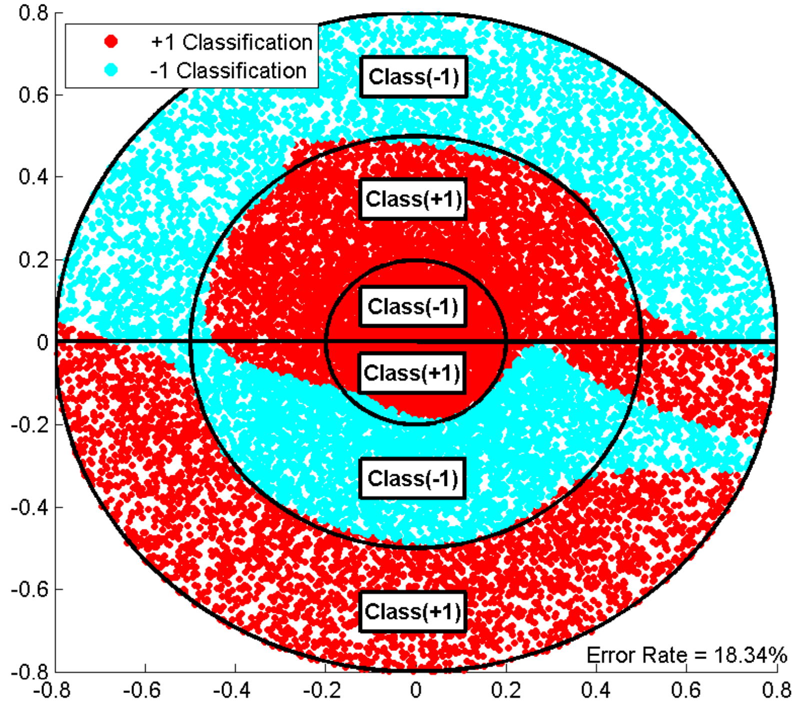

Figure 3.6: Result of classification on 20000 samples of test data using an MLP trained with 4000 epochs.

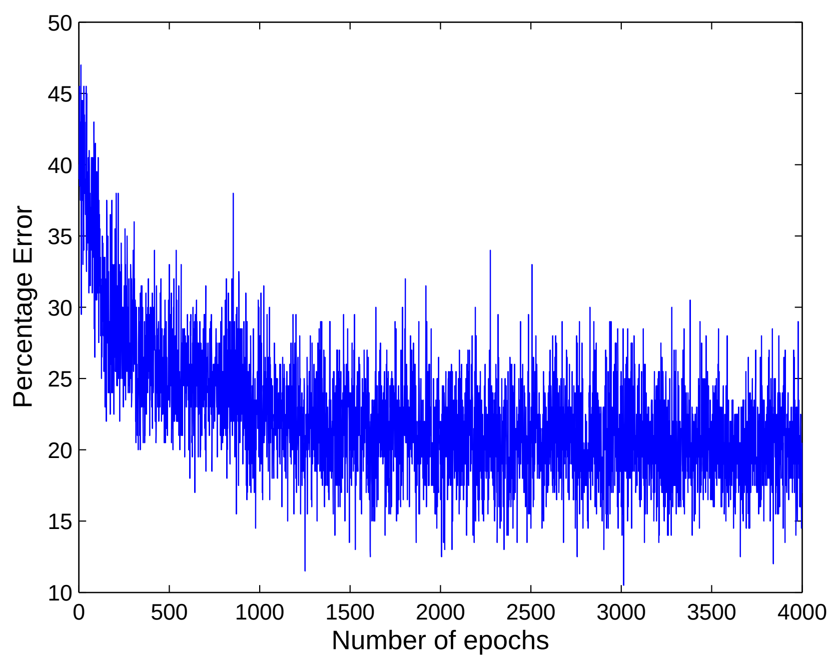

Figure 3.7: Learning rate curve of the MLP-EKF network using 4000 epochs.

Chapter 4

Discussion & Conclusion

After observing the results from Section 3, it is obvious that the SVM implementation is much better than the MLP-EKF implementation. By analyzing Figure 3.4, it is evident that the greater the C value is, the better the classifier performs. Large C values generally mean that the user trusts the quality of the training data used, where as small values of C mean that the training data is considered noisy and unreliable [2]. For the purpose of this experiment, the data that was generated was recorded without any noise which therefore supports the theory of increasing performance with an increasing value of C. Overall, the SVM implementation can be considered an excellent choice for nonlinear classification tasks.

The MLP-EKF algorithm is an efficient and power full tool for classification tasks, if the proper precautions are taken. A couple reasons for the extremely poor classification (Figure 3.7), are the improper initialization and the inadequate preprocessing of data. In the initialization stage, the state covariance matrix (P), process noise covariance matrix (Q), measurement noise covariance matrix (R) and the MLP parameter vector (θ) need to be properly initialized in order to train the MLP correctly. In this experiment, the state, process and measurement covariance matrices, were not properly tuned, which could have caused increased learning times and overall classification error.

Although initialization can dictate the course of learning for the MLP, the preprocessing stage could serve of great importance. In order to successfully preprocess the training data, the mean must be removed, followed by decorrelation and finally covariance equalization. In the algorithm show in A.1 the data was modified by removing the mean, but decorrelation and covariance equalization were not carried out. If these techniques were properly incorporated, then the classification errors would not be as high. In addition, the data could further be processed by introducing a nonlinear classification such as converting data from cartesian coordinates to polar coordinates. An example is shown in Figure 4.1, which is the polar form of Figure 2.2. By converting to polar coordinates, the classes become linearly separable, which can then increase the performance of both algorithms.

Overall, both algorithms should perform relatively well when the proper precautions are taken. Incorrect initialization can result in decreased performance as well as slow learning rates. As well, proper preprocessing is essential to increasing convergence times and decreasing error rates. The SVM algorithm classified the testing data with minimal error, while the MLP-EKF algorithm classified with an 40% error on average. With the use of better preprocessing techniques, proper

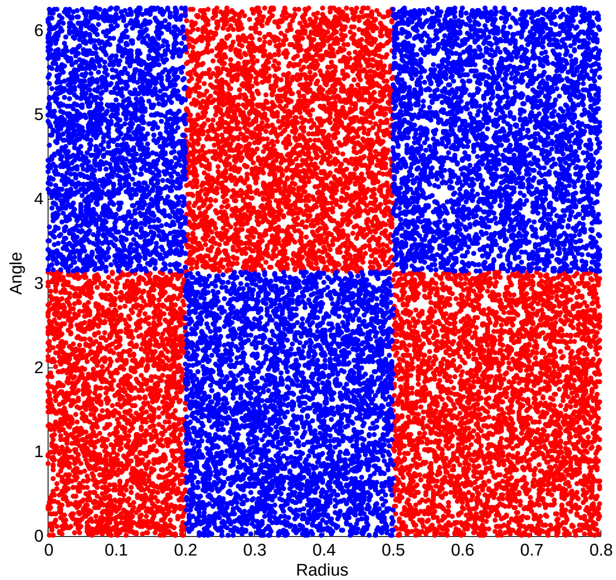

Figure 4.1: Polar coordinate version of Figure 2.2.

initialization and more efficient implementations, the MLP-EKF has the potential to outperform the SVM algorithm.

Appendix A

MATLAB Code

Listing A.1: MATLAB Script for simulating the classification problem and comparing the different learning machines involved

```
clear
close all
clc

%%                  CONSTANTS AND INITIALIZATIONS

%Constrain on the Lagrangian multipliers
C = 2500;

%1 means the SVM will be used, 0 means the MLPEKF is used
SVM_mode = 0;

%Colour selection for class (+1) and class(-1)
col(1,:) = [1 0 0];
col(2,:) = [0 1 1];

%Radii of the 3 concentric circles
r1 = 0.2;
r2 = 0.5;
r3 = 0.8;

%Size of the training and testing data
num_train_epoch = 5000;
num_train_samples = 200;

%                    MLP-EKF Initializations and Constants

count = 0; %Initializing counter for classification errors
scaler = 1; %For scaling the output value 'd' for the MLPEKF
nx = 2; %Input dimensionality
ny = 1; %Output dimensionality
nh1 = 4; %Number of nodes in the first layer
nh2 = 3; %Number of nodes in the second layer
```

```
 %Number of free parameters
ns = nh1*(nx+1) + nh2*(nh1+1) + ny*(nh2+1);

q = 0.001; %STD of process
r = 500; %STD of measurement
p = 100; %State covraiance magnitude

Q = q*eye(ns); %Initial process covariance
R = r*eye(num_train_samples); %Initial measurement covariance
P = diag(p*ones(1,ns)); %Initial state covraiance
m=4;
theta = sqrt(m)*randn(ns,1); %Initial guess of MLP parameters

%Tunable Activation function parameters
a = 1.7159;
b = 2/3;
%%                        GENERATING DATA
%generate the data for both training and testing
[training_data, training_data_colour] =...
    makedata(num_train_epoch, num_train_samples, r1,r2,r3, col);

[testing_data, testing_data_colour] =...
    makedata(num_train_epoch, num_train_samples, r1,r2,r3, col);

%%              Training and Testing using SVM Algorithm
if(SVM_mode == 1)

    %[svmStruct,error_perf,corr_perf] = SVM(training_data,...
    %testing_data,kernel,kernel_param_name, kernel_param_value,C)
    [svmStruct,error_SVM,corr_SVM]= SVM(training_data,...
                        testing_data,'rbf','RBF_Sigma', 0.5, C);

    %Preparing the test data
    test = [testing_data(:,num_train_epoch,1)...
            testing_data(:,num_train_epoch,2)];

    %Preparing the colouring scheme for the test data
    c(:,:) = testing_data_colour(:,num_train_epoch,:);

    %Record the indicies of matchin values in separate
    %vectors for each colour
    [r_index,b_index] = sep_colour(c,col);

    %Plotting the SVM classification results
    hold on
    figure;
    h = SVMPLOTTER(svmStruct,test,C);
    scatter(test(r_index,1),test(r_index,2),10,c(r_index,:),'filled');
    scatter(test(b_index,1),test(b_index,2),10,c(b_index,:),'filled');
```

```
    title(['SVM Classification for ',num2str(length(test(:,1)...
            )),' samples with C = ',num2str(C)],'FontSize',12);
    hold off
    xlim([-r3,r3]);
    ylim([-r3,r3]);

    %plot the learning curve
    plot(error_SVM)

else
    %%      Training and Testing using the MLP-EKF Algorithm

    for j = 1:num_train_epoch
        %Preparing the training set
        x = [training_data(:,j,1)'; training_data(:,j,2)'];
        x(1,:) = x(1,:) - mean(x(1,:));
        x(2,:) = x(2,:) - mean(x(2,:));
        y = scaler*training_data(:,j,3)';

        %Updating 'theta' using EKF method. Model is the MLP
        [theta,P,p]=MLPEKF(theta,P,x,y,Q,R,a,b);

        %Preparing the testing data
        x = [testing_data(:,j,1)';testing_data(:,j,2)'];
        x(1,:) = x(1,:) - mean(x(1,:));
        x(2,:) = x(2,:) - mean(x(2,:));
        y = scaler*testing_data(:,j,3)';

        %Extracting the weights from theta
        W1=reshape(theta(1:nh1*(nx+1)),nh1,[]);
        W2=reshape(theta(nh1*(nx+1)+(1:nh2*(nh1+1))),nh2,[]);
        W3=reshape(theta((nh1*(nx+1)+ nh2*(nh1+1)+1):end),ny,[]);

        %Evaluating the MLP using the updated 'theta' vector
        z = W3(:,1:nh2)*a*tanh(b*(W2(:,1:nh1)*a*tanh(b*(...
            W1(:,1:nx)*x+ W1(:,nx+ones(1,num_train_samples))))...
            + W2(:,nh1+ones(1,num_train_samples))))...
            + W3(:,nh2+ones(1,num_train_samples));

        %Calculating the error rate
        count=0;
        for i = 1:num_train_samples
            if(sign(z(1,i)) ~= y(1,i))
                count = count+1;
            end
            RMSE(j,1) = (mean((z-y).^2)).^0.5;
            MSE(j,1) = mean((z-y).^2);
        end

        %Plotting the classified test data
```

```
        if (j == 100)
            figure;
            c(:,:) = testing_data_colour(:,num_train_epoch,:);
            ff = sign(z); %Classify the output of the MLP {-1,1}

            %Create colouring scheme for classified data
            for u=1:num_train_samples
                if(ff(u)== 1)
                    c(u,:)= col(1,:); %class(+1)
                elseif(ff(u)== -1)
                    c(u,:)= col(2,:); %class(-1)
                elseif(ff(u)== 0)
                    c(u,:)= [0 0 0]; %boundary
                end
            end

        %Record the indicies in separate vectors for each colour
            [r_index,b_index] = sep_colour(c,col);

            hold on
            scatter(x(1,r_index),x(2,r_index),10,c(r_index,:),'filled');
            scatter(x(1,b_index),x(2,b_index),10,c(b_index,:),'filled');
            hold off
            xlim([-r3,r3]);
            ylim([-r3,r3]);
        end

        error_rate(j,1) = 100*count/num_train_samples;
         %for monitoring purposes
        percent_done = 100*j/num_train_epoch
    end

    %Plotting the error rate
    figure;
    plot(error_rate);
    xlabel('Number of epochs','FontSize',12);
    ylabel('Percentage Error','FontSize',12);
end
```

Listing A.2: MATLAB Script used to generate random data for training and testing

```
function [data,colour]=makedata(num_epoch,num_samples,r1,r2,r3,cc )
% Description: This function genereates data which is uniformly
% distributed on the image. 50% of the randomly distributed data
% is class(+1) and the remaining class(-1).
%
% Inputs:
%     num_epoch   = the number of epochs, scalar
%     num_samples = the number of samples per epoch, scalar
```

```
%     r1,r2,r3    = radii of the 3 concentric circles, scalar
%     cc          = the colours associated with each class,
%                     where cc(1,:) represents a RGB vector, 2x3
% Outputs:
%     data(:,:,1) = x-coordinates, num_samples x num_epoch
%     data(:,:,2) = y-coordinates, num_samples x num_epoch
%     data(:,:,3) = classification, num_samples x num_epoch
%     colour      = RGB vector (1x3) associated with every
%                       sample, num_samples x num_epoch

N1 = num_epoch;
N2 = num_samples;
for j=1:N1
    num_positive = 0;
    num_negative = 0;
    i = 1;
    while(num_positive < 0.5*N2 || num_negative < 0.5*N2)
        r = abs(r3*rand(1,1));
        theta = abs(2*pi*rand(1,1));
        %if on the boundary, it belongs to class 1 (grey)
        if (theta>0 && theta<pi && r<r1 && num_positive < 0.5*N2)
            %class0
            x1(i,j) = r;
            x2(i,j) = theta;
            d(i,j) = -1;
            num_positive = num_positive + 1;
            i = i+1;
        elseif (theta>=pi && (theta<=2*pi || theta ==0) &&...
                r<=r1 && num_negative < 0.5*N2)
            %class1
            x1(i,j) = r;
            x2(i,j) = theta;
            d(i,j) = 1;
            num_negative = num_negative + 1;
            i = i+1;
        elseif (theta>pi && theta<2*pi && r<r2 && r>r1 &&...
                num_positive < 0.5*N2)
            %class0
            x1(i,j) = r;
            x2(i,j) = theta;
            d(i,j) = -1;
            num_positive = num_positive + 1;
            i = i+1;
        elseif (theta>=0 && theta <=pi && r<=r2 && r>=r1 &&...
                num_negative < 0.5*N2)
            %class1
            x1(i,j) = r;
            x2(i,j) = theta;
            d(i,j) = 1;
            num_negative = num_negative + 1;
```

```
            i = i+1;
        elseif (theta>0 && theta <pi && r<r3 && r>r2 &&...
                num_positive < 0.5*N2)
            %class0
            x1(i,j) = r;
            x2(i,j) = theta;
            d(i,j) = -1;
            num_positive = num_positive + 1;
            i = i+1;
        elseif (theta>=pi && (theta<=2*pi || theta==0) && ...
                r<=r3 && r>=r2 && num_negative < 0.5*N2)
            %class1
            x1(i,j) = r;
            x2(i,j) = theta;
            d(i,j) = 1;
            num_negative = num_negative + 1;
            i = i+1;
        end
    end
end

for j =1:N1
    for i = 1:N2
        if (d(i,j) == 1)
            colour(i,j,:) = cc(1,:);
        else
            colour(i,j,:) = cc(2,:);
        end
    end
end

%convert from polar to cartesian
data(:,:,1) = x1.*cos(x2); %x coordinate
data(:,:,2) = x1.*sin(x2); %y coordinate
data(:,:,3) = d;           %class
end
```

Listing A.3: MATLAB Script used create and train the MLP

```
function [svmStruct,error_perf, corr_perf] = SVM(training_data,...
    testing_data,kernel, kernel_param_name, kernel_param_value,C)

% Description: Trains a SVM using training data, and provides
%     classification results using testing data.
%
% Input:
%         training_data and testing_data: (:,:,1) contains x
%         coordinates, (:,:,2) contrains y coordinates and (:,:,1)
%         contains classification for corresponding coordinate.
```

```
%          Rows (n) represent number of samples and columns (m)
%          represent number of epochs.
%
%                  kernel: string containing the type of kernel.
%                          e.g 'rbf'
%        kernel_param_name: string containg parameter related to the
%                          kernel function. e.g 'RBF_Sigma'
%       kernel_param_value: the value of the parameter. e.g 0.5
%                        C: constrain on the Lagrangian multipliers
%
% Output:
%                svmStruct: a SVM structure created using the
%                          svmtrain function
%               error_perf: a num_training_epoch x 1 vector
%                          containing the percentage of incorrect
%                          classifications
%                corr_perf: a num_training_epoch x 1 vector
%                          containing the percentage of correct
%                          classifications

%Storing the number of epoches in a variable for later use
num_epoch = length(training_data(1,:,1));

%Setting the SMO optimization algorithm to have a high max
%iteration number
SMO_OptsStruct = svmsmoset('MaxIter', 60000000);

%A loop which adds data to existing data for every epoch and then
%trains and evaluates classification performance for the test
%data, for every epoch.
for i=1:num_epoch
    %Initializes the training and testing data for concatenation
    %in later iterations.
    if(i==1)
        %Initialize training data
        data = [training_data(:,i,1) training_data(:,i,2)];
        data_group = training_data(:,i,3);

        %Initialize testing data
        test = [testing_data(:,i,1) testing_data(:,i,2)];
        test_group = testing_data(:,i,3);

        %Train the SVM using the input parameters
        svmStruct = svmtrain(data,data_group,'Kernel_Function',...
            kernel, kernel_param_name,kernel_param_value,...
            'METHOD','SMO','SMO_Opts', SMO_OptsStruct,...
            'BoxConstraint', C,'showplot',true);


        %Classifies the test data using the SVM structure made by
```

```
        %svmtrain
        svmResult  = svmclassify( svmStruct,test,'showplot',false);

        %Evaluates the perfomance of the classifier on the test
        %data and stores correct and incorrect classification
        %rates into to vectors of the same length
        cp = classperf(test_group,svmResult);
        error_perf(i,1) = cp.ErrorRate;
        corr_perf(i,1) = cp.CorrectRate;

        %For monitoring purposes
        (i*100)/num_epoch

    %Concatenates new data with the old data and continues the
    %the training and classification of the test data.
    else
        %Prepares the NEW training data for concatenation
        new_data = [training_data(:,i,1) training_data(:,i,2)];
        new_data_group = training_data(:,i,3);

        %Prepares the NEW testing data for concatenation
        new_test = [testing_data(:,i,1) testing_data(:,i,2)];
        new_test_group = testing_data(:,i,3);

        %Concatenates NEW training data with OLD training data
        data = vertcat(data, new_data );
        data_group = vertcat(data_group,new_data_group);

        %Concatenates NEW testing data with OLD testing data
        test = vertcat(test,new_test);
        test_group = vertcat(test_group,new_test_group);

        %Train the SVM using the updated input parameters
        svmStruct = svmtrain(data,data_group,'Kernel_Function',...
            kernel, kernel_param_name,kernel_param_value,...
            'METHOD','SMO','SMO_Opts', SMO_OptsStruct,...
            'BoxConstraint', C,'showplot',false);

        %Classifies the updated test data using the SVM structure
        %made by svmtrain
        svmResult  = svmclassify( svmStruct,test,'showplot',false);

        %Evaluates the perfomance of the classifier on the
        %updated test data and stores correct and incorrect
        %classification rates into to vectors of the same length
        cp = classperf(test_group,svmResult);
        error_perf(i,1) = cp.ErrorRate;
        corr_perf(i,1) = cp.CorrectRate;

        %For monitoring purposes
```

```
        (i*100)/num_epoch
    end
end
end
```

Listing A.4: MATLAB Script used plot the results of SVM classification

```
function [ h ] = SVMPLOTTER(svmStruct,test,C)
% Description: Uses an SVM structure as an input to classify the
%  input test data
%
% Input:
%     svmStruct: Is a SVM stucture that was created using SVM train
%          test: Test data, num_training_epoch x
%                 num_training _samples x 3
%             C: The positive contraint parameter
% Output:
%             h: A figure handle
%Adapted and modified from the MATLAB biolearning toolbox
sample = test;
sampleOrig = sample;
if ~isempty(svmStruct.ScaleData)
    for c = 1:size(sample, 2)
        sample(:,c) = svmStruct.ScaleData.scaleFactor(c) * ...
            (sample(:,c) +  svmStruct.ScaleData.shift(c));
    end
end
groupnames = svmStruct.GroupNames;
[g,groupString] = grp2idx(groupnames);

classified = svmdecision(sample,svmStruct); %classifies the data

%plotting the results of the training and testing data
h = figure(1);
hAxis = svmStruct.FigureHandles{1};
hLines = svmStruct.FigureHandles{2};
hSV = svmStruct.FigureHandles{3};
[hAxis,hClassLines] = svmplotdata_rob(sampleOrig,classified,hAxis);
trainingString = strcat(cellstr(groupString),' (training)');
sampleString = strcat(cellstr(groupString),' (classified)');
% legend([hClassLines(1),hClassLines(2),hSV],...
%    {sampleString{1},...
%    sampleString{2},'Support Vectors'});
legend off
xlabel('x_{1}','FontSize',12);
ylabel('x_{2}','FontSize',12);
% title(['SVM Classification for ',num2str(length(test(:,1))),
%' samples with C = ',num2str(C)]);
end
```

```
function [out,f] = svmdecision(Xnew,svm_struct)
%SVMDECISION evaluates the SVM decision function

%   Copyright 2004-2006 The MathWorks, Inc.
%   $Revision: 1.1.12.4 $  $Date: 2006/06/16 20:07:18 $

sv = svm_struct.SupportVectors;
alphaHat = svm_struct.Alpha;
bias = svm_struct.Bias;
kfun = svm_struct.KernelFunction;
kfunargs = svm_struct.KernelFunctionArgs;

f = (feval(kfun,sv,Xnew,kfunargs{:})'*alphaHat(:)) + bias;
out = sign(f);
% points on the boundary are assigned to class 1
out(out==0) = 1;
end
```

Listing A.5: MATLAB Script used to set plotting parameters for SVMPLOTTER

```
function [hAxis,hLines] = svmplotdata_rob(x,group,theAxis)
% SVMPLOTDATA plots 2-D data in SVM functions

% Copyright 2004-2006 The MathWorks, Inc.

holdState = ishold;
if nargin == 2
    class1 = 'r.';
    class2 = 'g.';
else
    axes(theAxis);
    hold on;
    class1 = 'g.';
    class2 = 'g.';
end
Xp = x(group==1,:);
h1 = plot(Xp(:,1),Xp(:,2),class1,'LineWidth',2);
hAxis = get(h1,'parent');
hold on
Xn =  x(group==-1,:);
h2 = plot(Xn(:,1),Xn(:,2),class2,'LineWidth',2);
if isempty(hAxis)
    h1 =0;
    hAxis = get(h2,'parent');
end
if isempty(h2)
    h2 = 0;
end
```

```
%axis equal
drawnow
% reset hold state if it was off
if ~holdState
    hold off
end
hLines = [h1,h2];
end
```

Listing A.6: MATLAB Script used create and train the MLP

```
function [theta,P,e]=MLPEKF(theta,P,x,y,Q,R,a,b)
% MLPEKF     A function using the EKF to training a MLP NN
% [theta,P,z]=MLPEKF(theta,P,x,y,Q,R) searches the optimal
% parameters, theta of  a MLP NN based on a set of training
% data with input x and output y.
% Input:
%     theta: Initial guess of MLP NN parameter. The network
%              structure is determined by the number of parameters,
%              ns, the number of inputs (size of x),nx and the
%              number of output (size of y), ny. The equation of
%              the NN is:
%              y = W3 * tanh( W2 * tanh( W1 * x + b1 ) + b2) + b3,
%              and theta = [W1(:);b1;W2(:);b2;W3(:);b3].
%         P: The covariance of the initial theta. Needs to be
%              tuned to get good training performance.
%   x and y: Input and output data for training. For batch
%              training, x and y should be arranged in such a way
%              that each observation corresponds to a column.
%         Q: The virtual process covariance for theta, normally
%              set to very small values.
%         R: The measurement covariance, dependent on the noise
%              level of data, tunable.
%Modified version of Yi Cao's algorithm at
%Cranfield University, 02/01/2008
f=@(u)u; % dummy process function to update parameters
h=@(u)nn(u,x,size(y,1),a,b); % NN model
[theta,P]=ekf(f,theta,P,h,y(:),Q,R);% the EKF
e=h(theta);% returns trained model output

% The NN model. Modified from original scrpit to include a second
% hidden layer in its structure.
function y=nn(theta,x,ny,a,b)
nh1 = 4; %number of nodes in first hidden layer
nh2 = 3; %number of nodes in first hidden layer

[nx,N]=size(x); %[input dimensionality, number of samples]
ns=numel(theta);

```

```
%Extracting weights from theta
W1=reshape(theta(1:nh1*(nx+1)),nh1,[]);
W2=reshape(theta(nh1*(nx+1)+(1:nh2*(nh1+1))),nh2,[]);
W3=reshape(theta((nh1*(nx+1)+ nh2*(nh1+1)+1):end),ny,[]);

%The NN model
y = W3(:,1:nh2)*a*tanh(b*(W2(:,1:nh1)*a*tanh(b*(W1(:,1:nx)*x+...
    W1(:,nx+ones(1,N))))+ W2(:,nh1+ones(1,N))))...
    + W3(:,nh2+ones(1,N));

y=y(:);% correct vector orientation for EKF
```

Listing A.7: MATLAB Script describing EKF learning algorithm for training the MLP

```
function [x,P]=ekf(fstate,x,P,hmeas,z,Q,R)
% EKF   Extended Kalman Filter for nonlinear dynamic systems
% [x, P] = ekf(f,x,P,h,z,Q,R) returns state estimate, x and state
%              covariance, P for nonlinear dynamic system:
%            x_k+1 = f(x_k) + w_k
%            z_k   = h(x_k) + v_k
% where w ~ N(0,Q) meaning w is gaussian noise with covariance Q
%       v ~ N(0,R) meaning v is gaussian noise with covariance R
% Inputs:    f: function handle for f(x)
%            x: "a priori" state estimate
%            P: "a priori" estimated state covariance
%            h: fanction handle for h(x)
%            z: current measurement
%            Q: process noise covariance
%            R: measurement noise covariance
% Output:    x: "a posteriori" state estimate
%            P: "a posteriori" state covariance
%
% By Yi Cao at Cranfield University, 02/01/2008
[x1,A]=jaccsd(fstate,x); %nonlinear update and linearization
                         %at current state
P=A*P*A'+Q;              %partial update
[z1,H]=jaccsd(hmeas,x1); %nonlinear measurement and linearization
P12=P*H';                %cross covariance
% K=P12*inv(H*P12+R);    %Kalman filter gain
% x=x1+K*(z-z1);         %state estimate
% P=P-K*P12';            %state covariance matrix
R=chol(H*P12+R);         %Cholesky factorization
U=P12/R;            %K=U/R'; Faster because of back substitution
x=x1+U*(R'\(z-z1));      %Back substitution to get state update
P=P-U*U';           %Covariance update, U*U'=P12/R/R'*P12'=K*P12.

function [z,A]=jaccsd(fun,x)
% JACCSD Jacobian through complex step differentiation
% [z J] = jaccsd(f,x)
```

```
% z = f(x)
% J = f'(x)
z=fun(x);
n=numel(x);
m=numel(z);
A=zeros(m,n);
h=n*eps;
for k=1:n
    x1=x;
    x1(k)=x1(k)+h*i;
    A(:,k)=imag(fun(x1))/h;
end
```

Listing A.8: MATLAB Script used for recording indices of different classes

```
function [ r,b ] = sep_colour( array, col )
% Description: Creates 2 arrays, r and b,  that contain
% indicies of matching colour (classes)
%
% Input:
%  array: Is a colour array containing RGB vectors,
%                    num_train_samples x 3
%  col  : Matrix containing RGB vectors for the two classes, 2x3

[N,dummy] = size(array); %Just need to store number of rows
a=1;
d=1;
for i=1:N
    if (array(i,:) == col(1,:))
        r(a,1) = i;
        a = a+1;
    else
        b(d,1)=i;
        d = d+1;
    end
end
end
```

Bibliography

[1] S. Theodoridis and K. Koutroumbas. "Pattern Recognition", 4th Ed., London: Elsevier Inc., 2009.

[2] Simon Haykin. "Neural Networks and Learning Machines", 3rd Ed., New Jersey: Pearson Education, Inc., 2009.

[3] DTREG, "SVM Support Vector Machines," 2007. [Online]. Available: http://www.dtreg.com/svm.htm. [Accessed April 2, 2011].

[4] DTREG, "Multilayer Perceptron Neural Networks," 2007. [Online]. Available: http://www.dtreg.com/mlfn.htm. [Accessed April 2, 2011].

[5] C.J.C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition,"Data mining and Knowledge Discovery, vol.2, issue.2, pp. 121-167, June 1998.

[6] S.R. Gunn, "Support Vector Machines for Classification and Regression". Faculty of Engineering, Science and Mathematics School of Electronics and Computer Science; 1998.

[7] John L. Crassidis and John L. Junkins, "Optimal Estimation of Dynamic Systems", Florida: CRC Press, LLC., 2004.