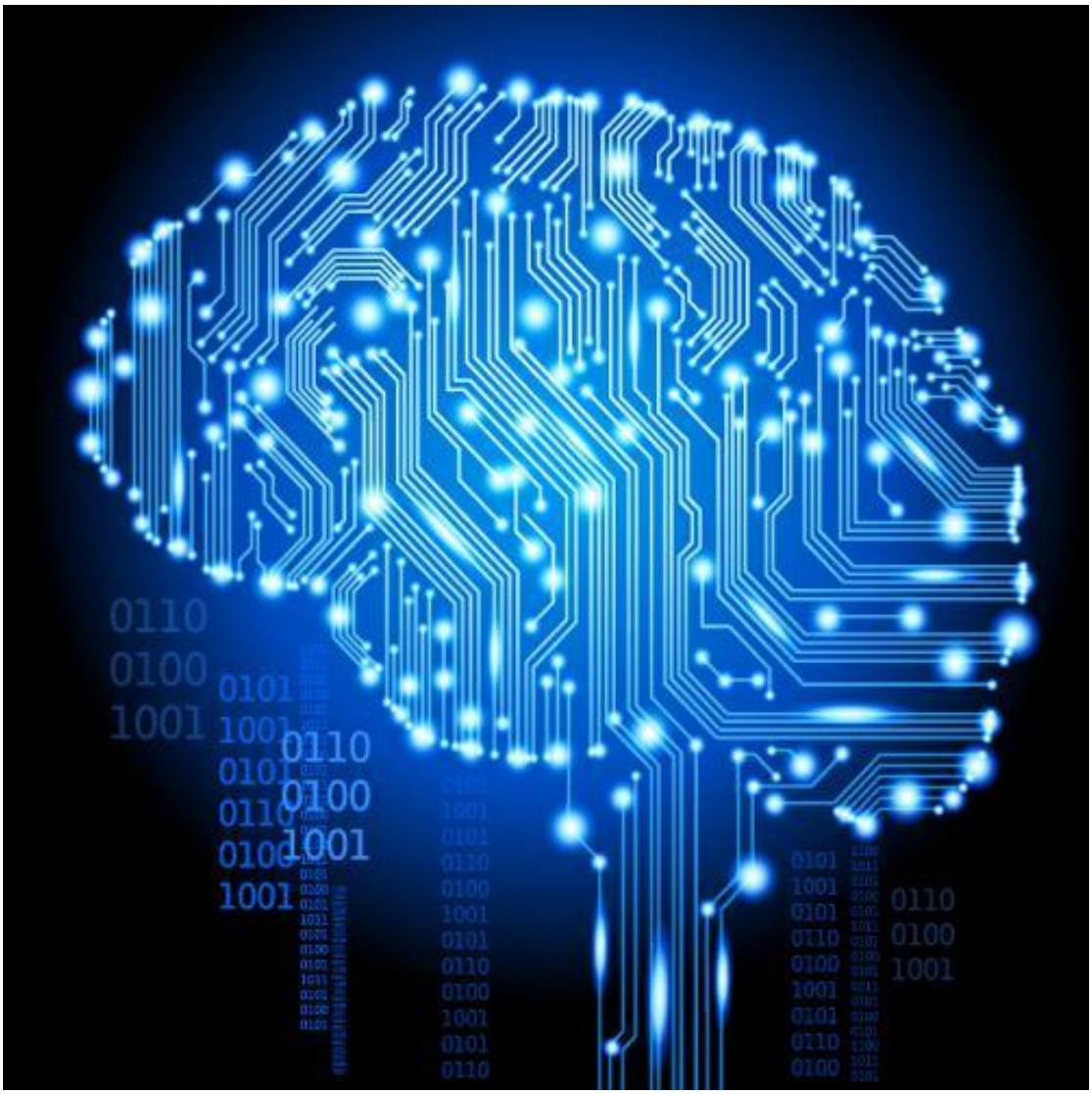
COMP3038 Machine Learning

Computer Based Coursework Manual – Autumn 2018



Lecturer:

Iman Yi Liao
Chong Siang Yew

Lab Assistant:

Lim Wei Xiang

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Table of Contents

1. Introduction

The purpose of this Computer-Based Coursework (LAB) is to provide you with hands-on experience in implementing and testing basic machine learning techniques. The techniques that will be assessed are Artificial Neural Networks (ANN), Decision Trees (DT), Support Vector Machines (SVMs), and Evaluating Hypotheses. Each of these techniques will be used in order to solve three problems: binary classification of smile/no smile expressions, multi-class classification of six universal basic facial expressions (anger, disgust, fear, happiness, sadness and surprise), and regression of the yaw-component of head-pose estimation. Background information on the six basic emotions will be given in section 3. You will be provided with a dataset to be used for each of the three problems.

2. Organization

2.1 Working Method

Implementation of the algorithms will be done in MATLAB. You will work in groups of 5 students (where possible). You are expected to work together on implementation of each machine learning technique and the related emotion recognizer. The groups and the lecture schedule are available on Moodle. The implementation will be either done from scratch (for Decision Trees, large aspects of Evaulating Hypotheses) or by using specialised Matlab toolboxes (ANNs and SVMs). After an assignment is completed, the Lab Assistant will evaluate the generated code of each group.

In addition, each group must hand in via Moodle a report of approximately 1,000 words plus results matrices and graphs, explaining details of the implementation process of each algorithm, for each of the three problems, along with comments on the acquired results. The generated code must also be included, as files in the zip package, with comments explaining the function of each important operation. Detailed deliverable details for every assignment will be described at the end of every section describing the assignment in question.

For each assignment, a zip or rar file with the name of the group should be submitted to Moodle. The zip should include (at least), the following items:

- The written report. It should be in PDF format.
- The corresponding trained models in a .mat file
- The code used to generate each of the assignments

Each group will be responsible to a great extend for the way in which the tasks and the reports are prepared and presented. These reports will provide feedback on the performance of the group as a whole. The individual performance of each group member will be evaluated during interviews of 5-10 minutes between the Lab Assistant and the whole group. The former will ask random group members about details of the delivered code and report. The interviews will take place during the first lab session following the delivery deadline of each assignment.

Note that, during the interview, Lab Assistants may ask you to test your trained models using held-out test data they will bring with them. You should therefore have a test script, which will receive the data as input and will return the predicted labels and a confusion matrix (see below).

2.2 Role of the Lab Assistant

The role of the Lab Assistant is to monitor the implementation of the assignments by the students. The Lab Assistant, however, will not make any substantive contribution to the implementation process. Final grading will be exclusively done by the convenor of the course, who will, nevertheless, ask for the recommendations of the Lab Assistants concerning the group progress.

2.3 Communication

Communication between the students and the Lab Assistant is very important, and will be done in labs during the lab sessions or via email using the addresses listed in the lecture slides.

Please mention your group number in any communication; this makes it easier for us to divide the work. In addition, students should visit the Moodle page of the course, which will contain links to the required data files as well as various MATLAB

functions needed to complete the assignments of this lab and also many other useful links and information.

2.4 Peer Assessment

After the three assignments are completed so-called *peer assessments* will be held. The members of each group will get the chance of passing anonymous feedback on the performance of their group members. This information will be used to make deviations in the individual grade.

The Lab Assistant will participate in the peer assessment as an extra group member, using your performance in the interviews to inform their judgment for the peer assessment.

Peer reviews will be disseminated and should be handed in through Moodle after the final coursework hand-in deadline.

2.5 Time Management

In total, there are 3 assessed assignments to be completed. As mentioned before, after the completion of each assignment a 1,000 word report must be handed in and a discussion between the Lab Assistants and the group members will take place. The deadlines for handing in assignments can be found on Moodle.

Missing any of the deadlines above will reflect on the final lab grade. Late submissions will result in 5% penalty per day. Reports will have to be handed in through Moodle. Only one report needs to be submitted per group. The Report should be accompanied by your source code, so hand in a single .zip archive.

2.6 Grading

$$m_g = \frac{1}{3}\sum_{l=1}^{3} m_l$$

$$m_i = \alpha_i * m_g$$

d the reports are handed in, every group peer assessment the final grade can be e expect each group member to actively gorithms, the reports and the interviews ch individual assignment will be graded eloped and the report that was delivered. will be a product of the group grade m_g bution \alpha_i to the group progress,

$$m_g = \frac{1}{3}\sum_{l=1}^{3} m_l$$

$$m_i = \alpha_i * m_g$$

Under the constraint that:

$$\frac{1}{n}\sum_{i}^{n}\alpha_i = 1$$

where m_l is the mark for lab l, and n the number of students in your group.

Attendance to the lab is mandatory and accounts for 30% of the **final grade for the Machine Learning Course**. Failure to attend the lab due to medical or other extenuating circumstances should be registered through the school office as a request for ECF.

2.7 Outline of the manual

The remainder of this lab manual is organized as follows: in section 3 an introduction on the six universal basic emotions will be given. In section 4, an introduction on MATLAB fundamentals is given. Section 5 will give an introduction to basic system-evaluation concepts, like K-fold cross-validation, confusion matrices, recall and precision rates. Section 6 gives some introductory toy problems that are optional for you to solve. Sections 7, 8, and 10 deal with the four machine learning techniques that have to be implemented.

3. The Six Universal Basic Emotions

One of the great challenges of our times in computer science research is the automatic recognition of human facial expressions. Machines capable of performing this task have many applications in areas as diverse as behavioural sciences, security, medicine, gaming and human-computer interaction (HCI). For instance, we use our facial expressions to synchronize a conversation, to show how we feel and to signal agreement, denial, understanding or confusion, to name just a few. Because humans communicate in a far more natural way with each other than they do with machines, it is a logical step to design machines that can emulate inter-human interaction in order to come to the same natural interaction between man and machine. To do so, machines should be able to detect and understand our facial expressions, as they are an essential part of inter-human communication.

Following early work by Charles Darwin, in the late 1970s American psychologist Paul Ekman confirmed the notion of six universal basic emotions intended to represent the most common high-level facial expression states, which include happiness, sadness, fear, sad, disgust and surprise (Figure 1).

Figure 1. – Paul Ekman's universal six basic emotions

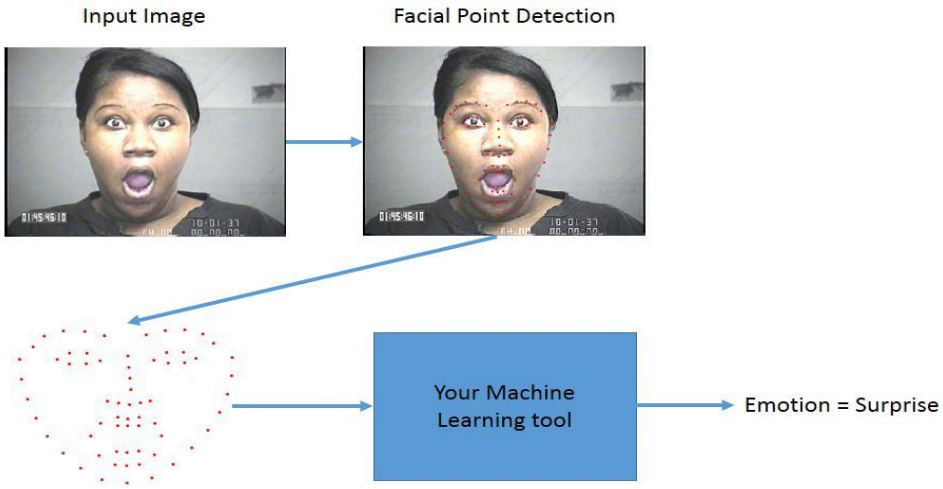
Thanks to its simplicity and relatively universal applicability, today this notion has become one of the most famous and widely used ways of classifying human facial expressions in a variety of science applications. Computer science is no exception, so nearly all early attempts to automatically detect facial expressions were based on Ekman's basic emotions.

Despite their popularity, basic emotions are not by any means intended to describe all possible facial expressions people are able to express, number of which exceeds 10 thousand. Covering the most basic, common subset of facial expressions also makes them suffer from a variety of cultural differences, in particular when comparing how Western and Eastern people express their emotions. To address this problem, later in 1978 Paul Ekman in collaboration with Wallace Friesen introduced a much more sophisticated notion of low-level facial expression representation called Facial Action

Coding System (FACS). The main components of FACS are so-called Action Units (AUs) or facial muscle activation units. With FACS every possible facial expression thus could be described as a combination of AUs. Though AU detection in general results in a richer encoding of a person's emotional state and covers a wider range of facial expressions (such as different types of smiles) it is also much harder to automatically detect AUs than basic emotions. Therefore, for the purposes of this lab we will only consider detection of basic emotions.

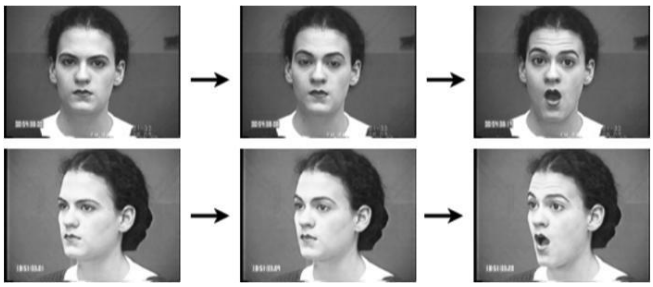
3.1 Goal

The goal of this coursework is to allow you to implement different Machine Learning techniques to predict emotions from a set of facial point locations. You will be provided with a tool that locates these points, and will be able to call specific methods of the tool to ultimately build your own "emotion tracking system".



3.3 Data

The data provided to you is the result of tracking face images with the iCCR face tracker, which tracks 66 facial points. For the binary problem, these images come from the MMI database, which is labelled for the presence of a smile. For the multi-class problem, they come from the CK+ database, which contains posed expressions, and that has been labelled with the 6 basic emotions described above.



The data for this lab will be provided in the form of .mat files which can be directly loaded in MATLAB using the load() function. On loading the file you will get the following:

- A matrix of size $n \times 132$, where n is the total number of examples (images) and 132 the dimensionality of the features (66 points with x and y coordinates).
- A vector y of length n, containing the labels of the corresponding examples. For smile detection, 1 indicates there is a smile. For the emotion problem, these labels are numbered from 1 to 6, and correspond to the emotions anger, disgust, fear, happiness, sadness and surprise respectively. For the head-pose estimation, it's the yaw angle in degrees.

In addition, we will provide functions that map emotion labels (numbers 1 to 6) to actual emotions (anger, disgust, fear, happiness, sadness, surprise) and back. These files are called *emolab2str.m* and *str2emolab.m* respectively.

Apart from the above data, we also provide the following toy datasets to be used only for the first two introductory lab sessions:

- Forest cover type dataset in cov1.mat
- OldFaithful dataset in OldFaithful.mat
- Texas temperature dataset in texas_temp.mat

Please note that the above 3 datasets will be used only for the introductory exercises of the first two lab sessions.

4. System Evaluation

In this section, the basic system evaluation concepts that will be used throughout this tutorial are given. These include:

- K-fold Cross Validation
- The Confusion Matrix
- Recall and Precision Rates
- The F_α-measure

a) Cross Validation

The concept of cross-validation is closely related to overfitting, a concept very important to machine learning. Usually a learning algorithm is trained using some set of training examples, i.e., exemplary situations for which the desired output is known. The learner is assumed to reach a state where it will also be able to predict the correct output for other examples, thus generalizing to situations not presented during training. However, in cases where learning was performed for too long or where training examples are not representative of all situations that can be encountered, the learner may adjust to very specific random features of the training data that have no causal relation to the target function. In the process of overfitting, the performance on the training examples still increases while the performance on unseen data becomes worse. An example of overfitting can be seen in Fig.3. The point where the error on the test set increases is the point where overfitting occurs.

In order to avoid overfitting, it is necessary to use additional techniques, one of which is cross-validation. Cross-validation is the statistical practice of partitioning a sample of data into subsets such that the analysis is initially performed on a single subset, while the other subset(s) are retained for subsequent use in confirming and validating the initial analysis. The initial subset of data is called the *training set*; the other subset(s) are called *validation* or *testing sets*.

The process of cross-validation is schematically depicted in Fig.4, where the initial dataset is split *N* times (*N-fold* cross validation) in order to give *N* error estimates. The final error will be the average of these *N* estimates, as shown in the figure.

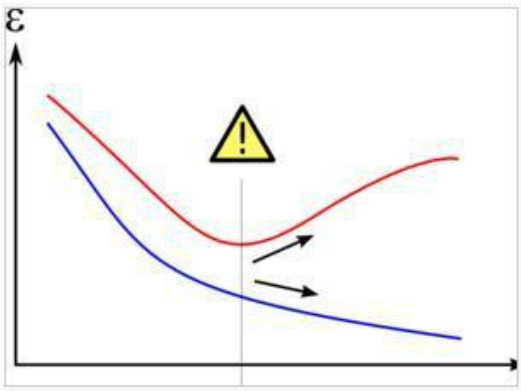


Fig.3: Overfitting/Overtraining in supervised learning (e.g. in a neural network). Training error is shown in blue, validation error in red. If the validation error increases while the training error steadily decreases then a situation of overfitting may have occurred.

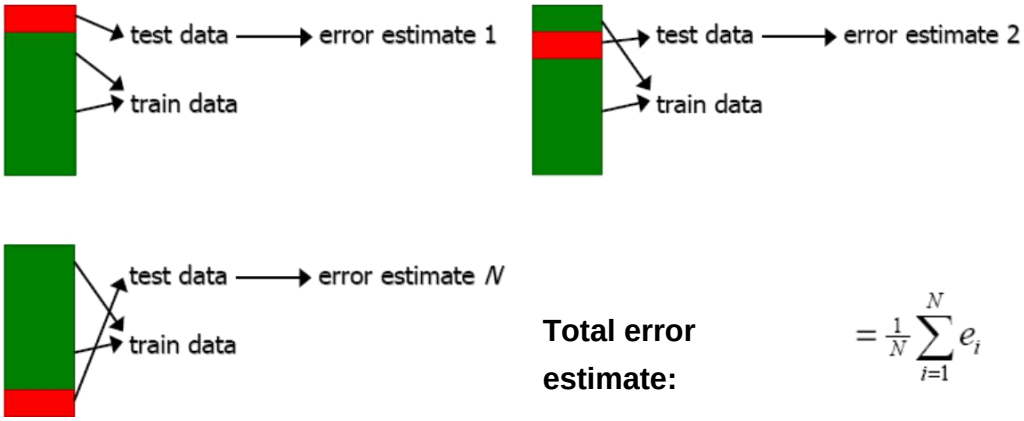


Fig.4: *N-fold* cross validation process. The data is split each time into training and testing datasets. The final prediction error is the mean average error.

b) Confusion Matrix

A confusion matrix is a visualization tool typically used to present the results attained by a learner. Each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class. One benefit of a confusion matrix is that it is easy to see if the system is confusing two classes (i.e. commonly mislabelling one as other). In the example confusion matrix below (Table 2), of the 8 actual cats, the system predicted that three were dogs, and of the six dogs, it predicted that one was a rabbit and two were cats. We can see from the matrix that the system in question has trouble distinguishing between cats and dogs, but can make the distinction between rabbits and other types of animals pretty well.

	Cat	Dog	Rabbit
Cat	5	3	0
Dog	2	3	1
Rabbit	0	2	11

Table 2: A simple confusion matrix

c) Recall and Precision Rates

Consider a binary classification problem, that is, containing only positive or negative examples, and a classifier that classifies these examples into two possible classes. A positive example assigned to the positive/negative class is called *True Positive/False Negative*. In the same way the *True Negative/False Positive* terms are defined. Based on this classification, the most obvious way to measure the performance of a classifier is to calculate the correct classification rate, defined as the sum of True Positives and True Negatives, divided by the total number of the examples.

There are cases, however, where the classification rate can be misleading. Consider the two following cases, depicted in the following tables:

Classifier	TP	TN	FP	FN	Recognition Rate
A	25	25	25	25	50%
B	37	37	13	13	74%

Classifier	TP	TN	FP	FN	Recognition Rate
A	25	75	75	25	50%
B	0	150	0	50	75%

Table 3: Two different classification scenarios. The top matrix depicts a classification problem using a balanced dataset (same number of positive and negative examples), while in the bottom matrix, the dataset is unbalanced.

It is clear, from the first table, that classifier B is better than A, since its classification rate is significantly larger and the number of positive and negative examples is the same. For the second table however, it is clear that, while classifier B correctly classifies the negative examples, it misses all the positive ones, in contrast to classifier A, whose classification performance is more balanced between the two classes. In order to overcome this ambiguity, and to be able to compare the two classifiers, the recall and precision rates are used instead.

Recall and Precision measure the quality of an information retrieval process, e.g., a classification process. Recall describes the completeness of the retrieval. It is defined as the portion of the positive examples retrieved by the process versus the total number of existing positive examples (including the ones not retrieved by the process). Precision describes the actual accuracy of the retrieval, and is defined as the portion of the positive examples that exist in the total number of examples retrieved. A schematic representation of the recall and precision rates is given in Fig.5. *A* represents the set of True Positives, *B* is the set of positive examples not retrieved, that is, False negatives and *C* is the set of negative examples that were wrongly classified as positives, that is, False Positives. Based on the recall and precision rates, we can justify if a classifier is better than another, i.e. if its recall and precision rates are significantly better.

Fig.5: Schematic representation of the recall and precision rates.

d) F_α measure

While recall and precision rates can be individually used to determine the quality of a classifier, it is often more convenient to have a single measure to do the same assessment. The F_α measure combines the recall and precision rates in a single equation :

$$F_\alpha = (1+\alpha)\frac{precision * recall}{\alpha * precision + recall},$$

where α defines how recall and precision will be weighted. In case recall and precision are evenly weighted then the F_1 measure is defined as follows:

$$F_1 = 2\frac{precision * recall}{precision + recall}$$

You are highly encouraged to write their own script to compute the Confusion Matrix, from which they would be able to compute the recall and precision rates, as well as the F1 score. You will be asked during lab interviews to produce a confusion matrix, as well as the F1 score and precision and recall rates, for a given test data. Failing to produce the Confusion Matrix and the scores will incur in a noticeable drop in the group's mark for that assessment.

e) Binary vs. Multi-class Classification

So far we have only talked about classifiers using datasets with only positive or negative examples. These classifiers are called binary classifiers, since the class labels can only take two values: ±1 (positive/negative). Many real- world problems, however, have more than two classes. In these cases we are talking about multi-class classification. To get M-class classifiers, it is common to construct a set of binary classifiers $f^1 \ldots f^M$, each trained to separate one class from the rest, and combine them for doing the multi-class classification according to the maximal output, that is, by taking:

$$\arg\max_{j=1\ldots M} g^j(x),$$

where $g^j(x) = \text{sgn}(f^j(x))$, and sgn() is the sign function. In case of a tie, i.e. two or more classifiers output +1, then the class is randomly chosen among the classes which were proposed by the classifiers. This is called one versus the rest classification, where each separate classifier is trained on one particular class, the positive class, while all the examples that belong to the rest of the classes are pooled together in order to form the negative class. One possible drawback of this method is that every classifier is trained on a small set of positive examples and a large set of negative ones, leading to an unbalanced training set, as described in section (c).

Pair wise classification trains a classifier for each possible pair of classes. For M classes, this results in (M-1)M/2 binary classifiers. This number is usually larger than the number of one-versus-the-rest classifiers; for instance, if M=10, we need to train

45 binary classifiers rather than 10 as in the method above. When we try to classify a test pattern, we evaluate all 45 binary classifiers, and classify according to which of the classes gets the highest number of votes. A vote for a given class is defined as a classifier putting the pattern into that class. The individual classifiers, however, are usually smaller in size than they would be in the one-versus-the-rest approach. This is for two reasons: first, the training sets are smaller, and second, the problems to be learned are usually easier, since the classes have less overlap.

5. Lab Session 1: Introduction to Matlab

This session is optional, can be done individually, and is not assessed.

MATLAB

In this part of the tutorial a brief introduction to some basic concepts of MATLAB will be given. You are strongly encouraged though to extensively use the MATLAB help files, accessible via the main MATLAB window. Type "doc" on the MATLAB command line to open the help browser. MATLAB blogs also provide useful information about how to program in MATLAB (http://blogs.mathworks.com).

a) Vectors and Arrays

A vector in MATLAB can be easily created by entering each element between brackets and assigning it to a variable, e.g.:

```
a = [1 2 3 4 5 6 9 8 7]
```

Let's say you want to create a vector with elements between 0 and 20 evenly spaced in increments of 2:

```
t = 0:2:20
```

MATLAB will return:

```
t =
     0   2   4   6   8   10   12   14   16   18   20
```

Manipulating vectors is almost as easy as creating them. First, suppose you would like to add 2 to each of the elements in vector 'a'. The equation for that looks like:

```
b = a + 2

b =
     3   4   5   6   7   8   11   10   9
```

Now, suppose you would like to add two vectors. If the two vectors are the same length, it is easy. Simply add the two as shown below:

```
c = a + b

c =
     4   6   8   10   12   14   20   18   16
```

In case the vectors have different lengths, then an error message will be generated.

You can do a scalar multiplication of a vector (or matrix), which simply multiplies each element of the vector/matrix by that scalar:

```
c = 3

d = 2 * c

d =
     12   18   24   30   36   42   60   54   48
```

The inner product of two vectors is defined as the sum of the element-wise components of two vectors of equal length. This can be written as a = **b`c**. If **b** = [b1 b2 b3] and **c** = [c1 c2 c3] then the scalar a = sum([b1*c1 b2*c2 b3*c3]).

Entering matrices into MATLAB is the same as entering a vector, except each row of elements is separated by a semicolon (;) or a return:

```
B = [1 2 3 4;5 6 7 8;9 10 11 12]

     B =
           1    2    3     4
           5    6    7     8
           9   10   11    12

B = [ 1  2  3  4
      5  6  7  8

      9 10 11 12]

     B =
           1    2    3     4
           5    6    7     8
           9   10   11    12
```

Matrices in MATLAB can be manipulated in many ways. For example, you can find the transpose of a matrix using the apostrophe symbol:

```
C = B'

     C =
           1   5    9
           2   6   10
           3   7   11
           4   8   12
```

Now, you can multiply the two matrices B and C. Remember that, unlike multiplying scalars, order matters when multiplying matrices (i.e. matrix multiplication is **not** commutative). When multiplying two matrices B and C, each element i,j (row i, column j) of the returned matrix D is formed as the inner product of the i'th row of B and the j'th column of C. As such, if B is of size m1 x n1, and C of m2 x n1, D will be of size m1 x n2. This also means for the inner product, and therefore the matrix multiplication, to work, n1 and m2 must have the same value, i.e. the number of columns of B must be the same as the number of rows of C.

```
D = B * C

      D =
          30    70   110
          70   174   278
         110   278   446

D  = C * B

      D =
         107   122   137   152
         122   140   158   176
         137   158   179   200
```

```
   152   176   200   224
```

Another option for matrix manipulation is that you can multiply the corresponding elements of two matrices using the .* operator. The matrices must be the same size to do this, and multiplications will be between elements with the same index in both matrices.

```
E = [1 2;3 4]
F = [2 3;4 5]
G = E .* F
     E =
          1   2
          3   4
     F =
          2   3
          4   5
     G =
          2   6
         12  20
```

MATLAB also allows multidimensional arrays, that is, arrays with more than two subscripts. For example,

```
R = randn(3,4,5);
```

creates a 3-by-4-by-5 array with a total of 3x4x5 = 60 normally distributed random elements.

b) Cell arrays and structures

Cell arrays in MATLAB are multidimensional arrays whose elements are copies of other arrays. A cell array of empty matrices can be created with the `cell` function. But, more often, cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{}`. The curly braces are also used with subscripts to access the contents of various cells. For example

```
C = {A sum(A) prod(prod(A))}
```

produces a 1-by-3 cell array. There are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces, for example *C{1}* retrieves the first cell of the array. Second, cell arrays contain *copies* of other arrays, not *pointers* to those arrays. If you subsequently change `A`, nothing happens to `C`.

Three-dimensional arrays can be used to store a sequence of matrices of the *same* size. Cell arrays can be used to store sequences of matrices of *different* sizes. For example,

```
M = cell(8,1);
for n = 1:8
   M{n} = magic(n);
end
M
```

produces a sequence of magic squares of different order:

```
M =
```

```
[           1]
[ 2x2  double]
[ 3x3  double]
[ 4x4  double]
[ 5x5  double]
[ 6x6  double]
[ 7x7  double]
[ 8x8  double]
```

Structures are multidimensional MATLAB arrays with elements accessed by textual *field designators*. For example,

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

creates a scalar structure with three fields.

```
S =
     name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

Like everything else in MATLAB, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

```
S(2).name = 'Toni Miller';
S(2).score = 91;
S(2).grade = 'A-';
```

Or, an entire element can be added with a single statement.

```
S(3) = struct('name','Jerry Garcia',...
              'score',70,'grade','C')
```

Now the structure is large enough that only a summary is printed.

```
S =
1x3 struct array with fields:
    name
    score
    grade
```

There are several ways to reassemble the various fields into other MATLAB arrays. They are all based on the notation of a *comma separated list*. If you type

```
S.score
```

it is the same as typing

```
S(1).score, S(2).score, S(3).score
```

This is a comma separated list. Without any other punctuation, it is not very useful. It assigns the three scores, one at a time, to the default variable `ans` and dutifully prints out the result of each assignment. But when you enclose the expression in square brackets,

```
[S.score]
```

it is the same as

```
[S(1).score, S(2).score, S(3).score]
```

which produces a numeric row vector containing all of the scores.

```
 ans =
     83    91    70
```

Similarly, typing

```
S.name
```

just assigns the names, one at time, to `ans`. But enclosing the expression in curly braces,

```
{S.name}
```

creates a 1-by-3 cell array containing the three names.

```
ans =
    'Ed Plum'    'Toni Miller'    'Jerry Garcia'
```

And

```
char(S.name)
```

calls the `char` function with three arguments to create a character array from the `name` fields,

```
ans =
Ed Plum
Toni Miller
Jerry Garcia
```

c) Functions

To make life easier, MATLAB includes many standard functions. Each function is a block of code that accomplishes a specific task. MATLAB contains all of the standard functions such as sin, cos, log, exp, sqrt, as well as many others. Commonly used constants such as pi, and i or j for the square root of -1, are also incorporated into MATLAB.

```
sin(pi/4)

ans =

    0.7071
```

To determine the usage of any function, type `help [function name]` at the MATLAB command window.

MATLAB allows you to write your own functions with the *function* command. The basic syntax of a function is:

```
function [output1,output2] = filename(input1,input2,input3)
```

A function can input or output as many variables as are needed. Below is a simple example of what a function, `add.m`, might look like:

```
function [var3] = add(var1,var2)
%add is a function that adds two
numbers var3 = var1+var2;
```

If you save these three lines in a file called "add.m" in the MATLAB directory, then you can use it by typing at the command line:

```
y = add(3,8)
```

Obviously, most functions will be more complex than the one demonstrated here. This example just shows what the basic form looks like.

d) Loops

If you want to repeat some action in a predetermined way, you can use the *for* or *while loop*. All of the loop structures in MATLAB are started with a keyword such as "for", or "while" and they all end with the word "end".
The *for loop* is written around some set of statements, and you must tell MATLAB where to start and where to end. Basically, you give a vector in the "for" statement, and MATLAB will loop through for each value in the vector: For example, a simple loop will go around four times each time changing a loop variable, *j*:

```
for j=1:4,
     j
 end
```

If you don't like the *for loop*, you can also use a *while loop*. The *while loop* repeats a sequence of commands as long as some condition is met. For example, the code that follows will print the value of the *j* variable until this is equal to 4:

```
j=0
while j<5
      j
      j=j+1;
end
```

You can find more information about *for loops* on http://blogs.mathworks.com/loren/2006/07/19/how-for-works/

e) Reading from files / Writing to files

Before we can read anything from a file, we need to open it via the *fopen* function. We tell MATLAB the name of the file, and it goes off to find it on the disk. If it can't find the file, it returns with an error; even if the file does exist, we might not be allowed to read from it. So, we need to check the value returned by *fopen* to make sure that all went well. A typical call looks like this:

```
fid = fopen(filename, 'r');
if (fid == -1)
  error('cannot open file for
reading'); end
```

There are two input arguments to *fopen*: the first is a string with the name of the file to open, and the second is a short string which indicates the operations we wish to undertake. The string 'r' means "we are going to read data which already exists in the file." We assign the result of *fopen* to the variable *fid*. This will be an integer, called the "file descriptor," which we can use later on to tell MATLAB where to look for input.
There are several ways to read data from a file we have just opened. In order to read binary data from the file, we can use the *fread* command as follows:

```
A = fread(fid, count)
```

where *fid* is given by *fopen* and *count* is the number of elements that we want to read. At the end of the *fread*, MATLAB sets the file pointer to the next byte to be read. A subsequent *fread* will begin at the location of the file pointer. For reading multiple elements from the file a loop can be used in combination with *fread*.
If we want to read a whole line from the file we can use the *fgets* command. For multiple lines we can combine this command with a loop, e.g. :

```
while (done_yet == 0)

    line = fgets(fid);
    if (line == -1)
      done_yet = 1;
    end
end
```

Before we can write anything into a file, we need to open it via the *fopen* function. We tell MATLAB the name of the file, and give the second argument 'w', which stands for 'we are about to write data into this file'.

```
  fid = fopen(filename, 'w');
  if (fid == -1)
    error('cannot open file for
  writing'); end
```

When we open a file for reading, it's an error if the file doesn't exist. But when we open a file for writing, it's not an error: the file will be created if it doesn't exist. If the file does exist, all its contents will be destroyed, and replaced with the material we place into it via subsequent calls to *fprintf*. Be sure that you really do want to destroy an existing file before you call *fopen*!
There are several ways to write data to a file we have just opened. In order to write binary data from the file, we can use the *fwrite* command, whose syntax is exactly the same as *fread.* In the same way, for writing multiple elements to a file, *fwrite* can be combined with a loop.
If we want to write data in a formatted way, we can use the *fprintf function, e.g. :*

```
fprintf(fid, '%d %d %d \n', a, b, c);
```

which will write the values of *a,b,c* into the file with handle *fid*, leaving a space between them. The string *%d* specifies the precision in which the values will be written (single), while the string \n denotes the end of the line.
At the very end of the program, after all the data has been read or written, it is good to close a file:

```
  fclose(fid);
```

f) Avoiding “Divide by zero” warnings

In order to avoid “Divide by zero” warnings you can use the *eps* function. *eps(X)* is the positive distance from *abs(X)* to the next larger in magnitude floating point number of the same precision as X. For example if you wish to divide A by B, but B can sometimes be zero which will return *Inf* and it may cause errors in your program, then use *eps* as shown:

C = A / B; % If B is 0 then C is *Inf*
C = A / (B + eps); % Even if B is 0 then C will just take a very large value and not *Inf*.

g) Profiler / Debugging

The *profiler* helps you optimize M-files by tracking their execution time. For each function in the M-file, profile records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. To open the *profiler* graphical user interface select HOME->Run and Time. So if the execution of your code is slow you can use the *profiler* to identify those lines of code that are slow to execute and improve them.

Another useful function that can be used for debugging is the *dbstop* function. It stops the execution of the program when a specific event happens. For example the commands

dbstop if error
dbstop if warning

stop execution when any M-file you subsequently run produces a run-time error/warning, putting MATLAB in debug mode, paused at the line that generated the error. See the MATLAB help for more details. Alternatively, you can use the graphical user interface to define the events that have to take place in order to stop the program. Just select EDITOR menu ->Breakpoints->ERROR HANDLING Stop onErrors/Warnings.

h) The face tracking system

The detection of emotions will be based upon the facial points, as shown above. When we want to train and test our methods, we typically use the annotated points (when possible!). However, we aim at Machine Learning techniques to be used in general scenarios, and therefore we need to have all the tools needed to use them. In this lab, we will use a built-in system that estimates the facial points from the input images. You will be able to use this system to estimate emotions from the points!

The face tracking system can be downloaded from www.cs.nott.ac.uk/~psxes1/download_iccr_v1.php. You can use the track_G53MLE.m function provided with the lab code on Moodle to predict the points, and will find the following code (l.118)

```
if ~isempty( pts )
 registered_pts = register_points( [] , pts , model.shModel.s0 , 1 );

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% MACHINE LEARNING STUDENTS CODE!!!!%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

end
```

You can include your machine learning tool inside this box and try it with your own videos or your webcam.

6. Lab Session 2: Introductory Exercises

This session is optional, can be done individually, and is not assessed.

Now that you have been introduced to working in MATLAB, you can try your hands at the following exercises. Please note that these exercises will not be evaluated and it is not expected that you complete each and every part of this exercise. Instead, pick one that takes your fancy. Attempting these exercises will help you to get a good grasp of working in MATLAB and some of the basic concepts of machine learning.

a) Classification using K-Nearest Neighbours

K-Nearest Neighbours is a classification algorithm which assigns a label to a feature *x* according to a majority vote taken over the labels of the K training instances with smallest distance to *x* (see Fig below).

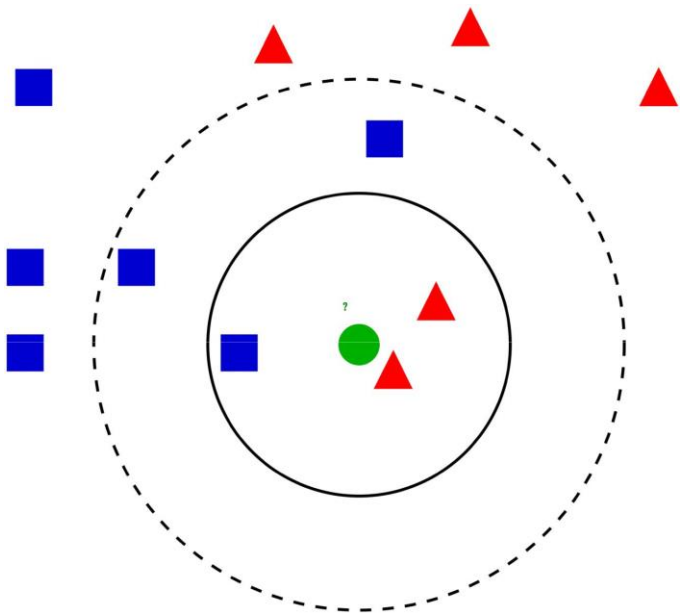


Fig. 6 k-Nearest Neighbours. If k = 3 (solid circle), then the new green data point will be assigned the label 'red triangle', as that's the majority label. If k = 5 on the other hand (dashed circle), it will be assigned the label 'blue square'.

Task: Write a function *label = knn(training_instances, training_labels, test_instance)* in MATLAB which takes as input the training instances, training labels and a test instance. The output should be a label for the test instance according to k-nearest neighbour algorithm. You can use the Euclidean distance as a distance measure for finding the nearest neighbours.

Use the first 1,000 or so data points from the forest cover type dataset in the file forest_covers.mat, included in the lab zip file, to test your function. Note that for efficiency reasons, forest_covers is stored as a sparse matrix. You can use it as you would a full matrix, but if you want to turn it into a 'normal' dataset, you could do:

```
load("forest_covers.mat");
x_train = full(training_vectors(1:1000, :));
y_train = training_labels(1:1000);
```

Additional Exercises:

1. Think of a reasonable performance measure, and program this from first principles
2. What is the dimensionality of the forest_covers feature vectors?
3. What data should you test on to measure the generalisation error?
4. Try different values of k and measure their performance

b) Clustering using K-Means

K-Means is a clustering algorithm that is used for finding K clusters in a given set of data points where each data point belongs to a cluster with the nearest mean. It basically consists of an iterative procedure consisting of the following steps:

1. Initialise cluster centres $\boldsymbol{\mu_k}$ randomly.
2. Assign each point to one of the *k* clusters based on their distance to the current centres $\boldsymbol{\mu_k}$.
3. Update the cluster centre locations by calculating the new mean of the points assigned to each cluster.
4. Repeat steps 2 and 3 until convergence.

You can look up k-means on line for more details.

Task: Write a function *cluster_labels = k_means(data_points)* from first princples, which takes as input a set of data points and outputs a label for each point according to the cluster to which it belongs. Use the dataset from the file OldFaithful.mat to test your function.

Additional Exercises:

1. Write a function to visualise your end result
2. Write a function to visualise the intermediate results after each iteration of the algorithm
3. Try cluster centres in the range 2:5, and visualise the end results for each
4. Try different runs of the algorithm with the same number of clusters. Are the results always the same? Explain why.

c) Linear regression using a first order polynomial model

Regression is used for modelling the relationship between a response variable and one or more predictor variables. In linear regression, the response variable is modelled as a linear combination of the model parameters and the predictor variables.
Task: Write a function model_parameters = learnRegressionModel(predictions, responses) , where predictions is a matrix containing the observed values for predictor variables in each row and responses is a column vector containing the corresponding values for response variables. The function should return a vector containing the model parameters. Learning the model parameters requires the minimization of the least squares objective function. Write the above function first by

1. Using brute force approach, and then
2. Using gradient descent method.

Use the data from the file texas _temp.mat to test your function by learning a regression model to predict the temperature (response variable) from the predictor variables (latitude, elevation and longitude).

7. Lab Sessions 3 & 4 (Assignment 1): Artificial Neural Networks

This is the first of the assessed lab sessions. From here on, work should be done as a group.

An Artificial Neural Network (ANN) or simply Neural Network is a network of interconnected neurons (see Fig. 7) that is able to learn non-linear real-valued, discrete-valued or vector-valued functions from examples. They are inspired by the observation that biological learning systems are composed of very complex webs of neurons. Every neuron has a number of real-valued inputs, which can be the outputs of other neurons, and uses a mathematical function called the transfer function to compute one real-valued output. There are three types of neurons which are most commonly used: the *perceptron*, which takes a number of real valued inputs and computes a binary output as a linear combination of the inputs, the *linear unit* which takes a number of real valued inputs and computes a continuous output as a linear combination of the inputs, and the *sigmoid unit* which takes a number of real valued inputs and computes a continuous output as a non-linear, sigmoid function of the inputs.
The way the various neurons are connected is called the network topology. The neurons are interconnected with each other by synapses, the weights of which define the output of the neuron. In the training phase, it is these weights that need to be learned. Together with the topology and the type of neurons the weights completely define the ANN.
The procedure used to update the weights in order to minimize the mistakes made by the classifier is called the training rule, or learning law. In Tom Mitchell's "Machine Learning" book the *backpropagation* method is described. In this tutorial we will experiment with different versions of this training rule.

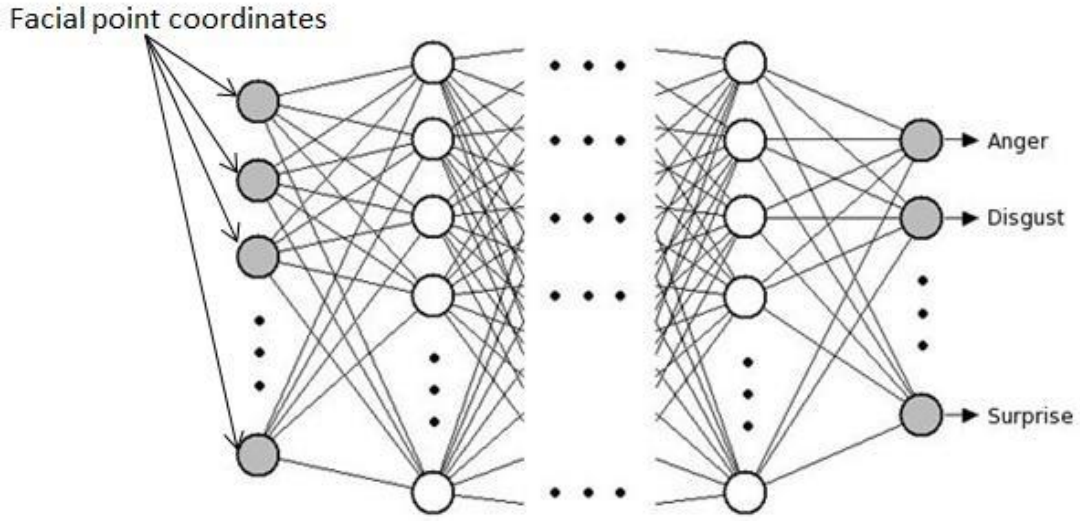


Figure 7.Artificial Neural Network with facial point coordinates as inputs and emotions as outputs

a) Implementation

To complete assignment 2, you will work with the MATLAB Neural Networks toolbox. The command *nntoo*l will open a dialog box where you can define the topology of your network, the data you want to use, the transfer function of the neurons, the learning law, and the function you want to use to measure the performance of the network. The transfer function can be specified per hidden layer, while the learning law and the performance measure are for the entire network. For the cross-validation evaluation of the ANNs you cannot use this GUI however and you should write your own cross-validation script using the toolbox's command line functions. It is recommended that you read through the excellent MATLAB help section on Neural Network Design before starting with this assignment. For this assignment we will be building feed-forward backpropagation neural networks.

a1. Create data

Make sure your data is loaded in the workspace using the *load* command. For the Neural Networks implementation of MATLAB, the features (i.e. facial point coordinates) should be arranged such that every column represents one sample. The label data depends on which of the three tasks you're learning: binary smile data, multiclass emotion data, or regression data.

For the binary smile data, change directory (cd) to the relevant assessed data folder, and load the facial points (features), and binary labels (smile/no smile):

```
load('facialPoints.mat');
load('labels.mat');
```

For the multiclass emotion problem, labels should be arranged in a matrix so that again every column represents the label of one sample. Every row of this matrix corresponds to the target values of the network's output nodes. Since every sample can have only one emotion label, only one element per column can have a value '1'. The row number that corresponds to the active emotion label for that sample should contain a value of '1', all other rows of that column should contain the value '0'. So a sample with the label 'happiness', a value of 4 in the data returned by *load*, would be represented by the vector $[0\ 0\ 0\ 1\ 0\ 0]^T$. Load the emotion data using the following command:

```
load('emotions_data.mat');
```

For the regression data, we will use the 6^{th} label of every data point for all assessed problems. Change directory to the headpose directory and do:

```
load('facialPoints.mat');
load('headpose.mat');
labels = pose(:,6);
```

Note that the format of the facial points for the regression problem is [nrPoints x 2 x nrFrames]. You will need to reshape your data so that it has the required form of [nrDataPoints x nrDimensions].

a2. Create network using nntool

Make sure the data is in the correct format, as specified above. Next, run the *nntool* command. This will open a Graphical User Interface (GUI), see Fig. 7.

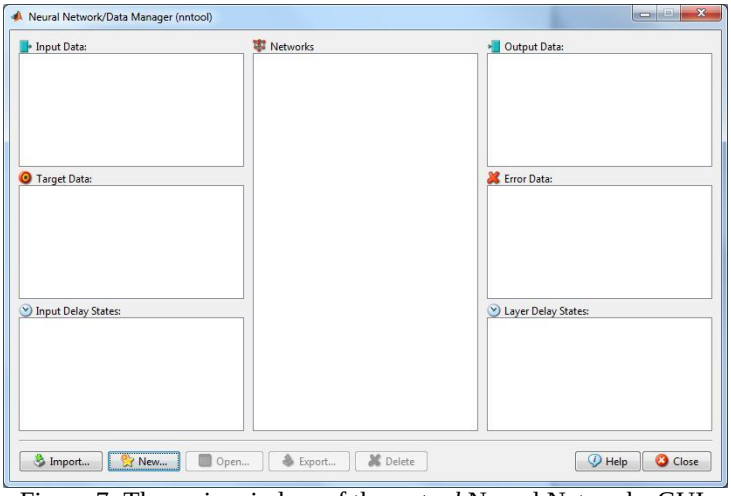


Figure 7: The main window of the *nntool* Neural Networks GUI.

Import the training data as inputs and as targets using the 'import' button. Make sure you import the facial point coordinates as 'inputs' and the emotion labels as 'targets'. Next create a new network by clicking the ‘New…’ button. This will bring up the ‘Create Network or Data’ window, see Fig. 8.

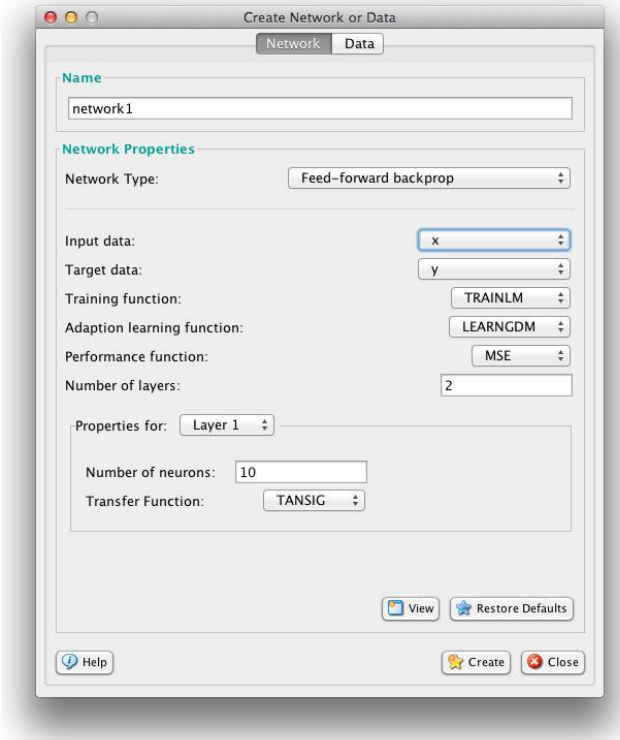


Figure 8: The ‘Create Network or Data’ window.

Select *Feed forward backprop* from the network type list. Choose your features as the input data and your labels as the target data. The training function should be one of the backpropagation algorithms. If you want to change the number of layers, make

sure to hit 'enter' after changing the value in the text box. Please note that the last layer you define is the output layer and as such should have the same number of neurons as the number of categories in which you want to classify your data. Click 'Create' when you are done.
Next select the network you've just created in the main window of the *nntool* and press the 'Open …' button. This will bring the network visualisation window. Go to the 'Train' tab, where you can set the training parameters, see Fig. 9.

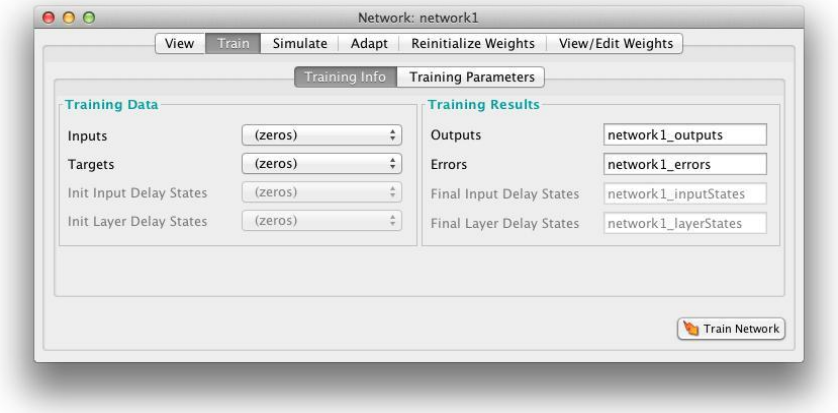


Figure 9: Setting the network's training parameters.

Specify the training input and targets. Be aware that some training functions are very slow, so don't start training with too many epochs (100 should be fine). You can set the value of the epochs in the 'Training Parameters' tab. Here you can also set how often the graphical output of training info is given. Set this value low, say every 5 or 10 epochs. When you feel confident that the NN is learning fast enough, you can increase the number of epochs if you wish. Note the training error. Toy with some of the training parameters. What are the influences of the learning rate and of the number of epochs used? These questions must be addressed in your report on the assignment 2.

a3. Evaluation

Now that you have a basic understanding of Neural Networks train a network using the emotion dataset from the previous assignment. You can use either the *nntool* or the command line functions as described below. Next, evaluate the neural networks using 10-fold cross validation. To do so, we will no longer use the graphical *nntool*. Instead, use the function *newff* to create a novel network. Read the help pages of *newff*, *network/train* and *network/sim*. Note that you can set the `NET.`*trainParam.show*, `NET.`*trainParam.epochs* and `NET.`*trainParam.goal* values of your network prior to training. The values of these parameters will influence classification performance and training time.

- `NET = newff(P,T,S,TF,BTF,BLF,PF,IPF,OPF,DDF)`
  takes
  - P - RxQ1 matrix of Q1 representative R-element input vectors.
  - T - SNxQ2 matrix of Q2 representative SN-element target vectors.
  - Si - Sizes of N-1 hidden layers, S1 to S(N-1), default is [].
  - TFi - Transfer function of ith layer. Default is *'tansig'* for hidden layers, and *'purelin'* for output layer.

- BTF - Backprop network training function, default is *'trainlm'*.
- BLF - Backprop weight/bias learning function, default is *'learngdm'*.
- PF - Performance function, default is *'mse'*.
- IPF - Row cell array of input processing functions. Default is {*'fixunknowns'*,*'remconstantrows'*,*'mapminmax'*}.
- OPF - Row cell array of output processing functions. Default is {*'remconstantrows'*,*'mapminmax'*}.
- DDF - Data division function, default is *'dividerand'*;

and returns a N layer feed-forward backprop network.

- `[NET, TR] = train(NET, X, T)`
  takes a network NET, input features X and targets T and returns the network after training it, and a training record TR.

- `t = sim(NET, X)`
  takes a network NET and input features X and returns the predicted labels t generated by the network.

For more information regarding the above methods please refer to the Matlab help. Make a 10-fold cross-validation evaluation of the neural networks using the parameters and learning rule that is optimal according to you. Evaluate each of the three problems (binary classification, multiclass classification, and regression). ***You will want to divide tasks between your group members.***

(***Hint***: 10-fold cross validation using neural networks will be used again in the last assignment so you may wish to write a function that takes as inputs the predicted labels t and the targets and returns a 6x6 confusion matrix).

b) Deliverables

For the completion of this part of the practical, the following have to be handed in:

1. MATLAB material, containing:
   - The resulting Artificial Neural Networks (one for each of the three ML problems), created by training on the whole dataset provided to you. Save the trained network with the command 'save'.
   - The code used to train the networks, including the loading and transforming of the data, initialisation of the NN and setting of network parameters.

2. Report of up to 1,000 words containing:
   - Explanation of the parameters chosen (e.g. topology, learning rule, learning rate, nr. of epochs, sizes,) and reports on the difficulties encountered.
   - Classification results per cross-validation fold for classification, and Root Means Square Error for the regression problem. This implies that you will have to write a small script that splits the given dataset into training and test sets.

- Explain what action you took to ensure generalisation of the network and overcome the problem of overfitting.
- Average cross validation classification results, that include:
    - Confusion matrix for the multiclass problem
      (***Hint***: you will be asked to produce confusion matrices in almost all the assignments so you may wish to write a general purpose function for computing a confusion matrix)
    - Average accuracy for the classification problems.
    - Average Root Mean Square Error for the regression problem
      (***Hint:*** you can derive them directly from the previously computed confusion matrix)

  This implies that you will have to write a small script that splits the given dataset into training and test sets.

8. Assignment 2: Decision Trees

This assignment will only require you to train and test on the **binary classification** data.

Decision trees are one of the simplest and yet most successful forms of learning algorithms. A decision tree takes as input an object or situation described by a set of attributes and returns a decision – the predicted value for the input. Classification is done by learning a function with discrete outputs based on the subset of data that has reached the node. A decision tree makes its decision by performing a sequence of tests, one per node on its path. Each internal node in the tree corresponds to a test of the value of one of the properties (for a monothetic tree), and the branches from the node are labelled with the possible values of the test. Problems where each test results in a binary outcome will generate binary trees, i.e. the branching factor of the tree will be 2. Each leaf node in the tree specifies the value to be returned if that leaf is reached. Decision tree representations seem to be very natural for humans, e.g. some "How To" manuals are written entirely as a single decision tree stretching over hundreds of pages. A pseudo code of the decision tree learning algorithm is depicted in Table 4.

function DECISION-TREE-LEARNING(features, labels) **returns** a decision tree
    inputs: *features*, set of training example features
        labels, target labels for the training examples

if all *examples* have the same label **then return** a leaf node with label = the label
else
    [*best_feature, best_threshold]* ← CHOOSE-ATTRIBUTE(*features*,*targets*) *tree* ← a new decision tree with root decision attribute *best*
    for each value v_i of *best* **do**
        add a branch to *tree* corresponding to *best* = v_i
        {$examples_i$, $targets_i$}← {elements of *examples* with *best* = v_i and corresponding targets}
        if $examples_i$ is empty **then return** a leaf node with label = MAJORITY- VALUE(*targets*)
        else *subtree* ← DECISION-TREE-LEARNING($examples_i$, *targets*)

    return *tree*

Table 4: Pseudo code for the decision tree learning algorithm.

The function MAJORITY-VALUE just returns the majority value of the example labels. The function CHOOSE-ATTRIBUTE measures how "good" each attribute (i.e. feature) in the set is. Several methods can be applied here. You should use the ID3 algorithm, based on information theory. Suppose the training set contains *p* positive and *n* negative examples. Then an estimate on the information contained in a correct answer is:

$$I\left(\frac{p}{p+n},\frac{n}{p+n}\right)=-\frac{p}{p+n}\log_2\frac{p}{p+n}-\frac{n}{p+n}\log_2\frac{n}{p+n}$$

Testing of any attribute A will divide the training set E into subsets $E_1,\ldots,E_\upsilon$ according to their values for A, where A can have υ distinct values (e.g. 2 for binary problems). Each subset E_i has p_i positive examples and n_i negative examples, so going along that branch, $I(p_i/(p_i+n_i), n_i/(p_i+n_i))$ bits of information will be needed to answer the question. So, on average, after testing attribute A we will need:

$$\mathrm{Re}\,mainder(A)=\sum_{i=1}^{\upsilon}\frac{p_i+n_i}{p+n}I\left(\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i}\right)$$

bits of information to classify the example. The information gain from the attribute test is the difference between the original information requirement and the new requirement:

$$Gain(A)=I\left(\frac{p}{p+n},\frac{n}{p+n}\right)-\mathrm{Re}\,mainder(A)$$

Finally, the attribute that is chosen is the one with the largest gain.
For more information on decision trees, see Chapter 8.1-8.3 of *Duda & Hart's "Pattern Classification"* book.

a) Implementation

The goal of the first assignment is to implement a decision tree algorithm using MATLAB, along with the accompanying functions, as described in the previous section. The inputs of the implemented algorithm should be the given set of examples, their respective target labels and the smile label that the resulting tree corresponds to.

a1. Create data

Make sure your data is loaded in the workspace using the *load* command. This function outputs an array *x*, which is an *N×132* array, where *N* is the total number of examples and 132 is the number of coordinates of facial points (or features/attributes) and an array *y* of dimensions *N×1*, containing the labels of the corresponding examples. These labels are numbered 0 or 1, indicating smile presence.

a2. Create decision tree

The output of your algorithm should be a single decision tree. You are expected to follow the algorithm shown in Table 4, and implement the ID3 algorithm for the attribute selection. The resulting tree must be a MATLAB structure (*struct*) with the following fields:

- *tree.op* : a label name (string) for the corresponding node (i.e. the name of the attribute that the node is testing). It must be an empty string for a leaf node.
- *tree.kids* : a cell array which will contain the subtrees that initiate from the corresponding node. Since the resulting tree will be binary, the size of this cell array must be 1x2, where the entries will contain the left and right subtrees respectively. This must be empty for a leaf node since a leaf has no kids, i.e. tree.kids = [].

- *tree.class* : a label for the returning class. This field can have the following possible values:
    - *0 - 1*: the classification of the examples (*negative-positive*, respectively), if it is the same for all, or as it is defined by the MAJORITY-VALUE function (in the case *attribs* is empty).
    - It must be empty for an internal node, since the tree returns a label only in a leaf node.
- *tree.attribute* : the attribute number tested at this node
- *tree.threshold* : the threshold for the attribute to decide which way to send data points

This tree structure is essential for the visualization of the resulting tree by using the ***DrawDecisionTree.m*** function, which is provided. Alternatively, you can choose a different tree structure, provided that you also provide a visualization function for this new structure.

a3. Evaluation

Now that you know the basic concepts of decision tree learning, you can use the clean dataset provided to train your tree, and visualise it using the DrawDecisionTree function. Then, evaluate your decision trees using 10-fold cross validation. You should expect that slightly different trees will be created per each fold, since the training data that you use each time will be slightly different. Use your resulting decision trees to classify your data in your test set. In order to evaluate a tree, you may wish to write a function that takes as inputs the tree you want to evaluate, the input samples and returns the outputs of the trees.

b) Deliverables

For the completion of this part of the tutorial, the following have to be handed in:

1. The code used to create the decision trees, including the loading and transforming of the data.

2. A report of up to 1,000 words containing the following:
    - The acquired decision tree.
    - Cross validation classification results, that include:
        - Recall and precision rates.
        - The F_1-measure derived from the recall and precision rates of the previous step.

      This implies that you will have to write a small script that splits the given dataset into training and test sets.

    Answer the following questions in your report:
    - Pruning is an important issue in trees. Explain what pruning does, and find the node(s) that would be pruned first for one of your learned trees. Explain the difference between the original and the pruned tree.
    - In this assignment, you trained a binary tree. Explain how you would use decision trees to learn to predict the multiclass problem of six-basic emotion recognition. Explain how to make decisions in leaf nodes, and how you would have to change your query search algorithm for any node.

9. Assignment 3: Support Vector Machines

In this assignment, you will train Support Vector Machines (SVMs) using different kernels, set hyper-parameters using inner-cross-fold validation, and you will compare the performance of SVMs with that of Decision Trees and ANNs in a statistically principled manner.

a. Train SVMs with the linear kernel

Use as the classifier a linear SVM, with the box-constraint C (which determines the importance of the slack-variables) set to be always one (i.e. no hyperparameters to tune). You have to build models for binary classification and regression. For binary classification, use the function:

```
Mdl = fitcsvm(X,Y, Name, Value);
```

to train a model, where X are the features and Y the labels of a two-class problem, and 'Name', 'Value' are variable name and value pairs to set. To set the kernel to linear, use the pair

```
'KernelFunction','linear'
```

and to set the box-constraint parameter to 1, use:

```
'BoxConstraint',1
```

So the full command would be:

```
Mdl = fitcsvm(X,Y, 'KernelFunction','linear', 'BoxConstraint',1);
```

For regression, use

```
Mdl = fitrsvm(X,Y, Name, Value)
```

to train a model. Note that for regression, you have to set the value of the parameter `'Epsilon'` to something sensible. Note that this is half of the error-insensitive tube diameter. Try a few different values of `'Epsilon'`. The model's performance is not important at this point.

b. Train SVMs with Gaussian RBF and Polynomial kernels

Use Matlab's built-in fitcsvm/fitrsvm to train models, now using RBF and Polynomial kernels. Write an inner-fold cross-validation routine for finding the optimal hyperparameters. For classification, these are C and the kernel parameters: `'KernelScale', sigma` for RBF and `'PolynomialOrder',` q for the polynomial kernel. For regression, you have to set `'Epsilon'` in addition. Do not use the automatic parameter optimisation methods (although you can compare against that if you wish to check if you're doing the right thing).

Report for each model you trained how many support vectors were selected, both in absolute terms and in terms of a % of the training data available.

c. Compare performance between methods

Perform 10-fold cross-validation on the binary classification data comparing ANNs, Decision Trees, and SVMs, and on the regression data comparing ANNs and SVMs (so trees only for the binary data). You must write your own 10-fold cross-validation, do not use built-in functions for this! Report results for linear, Gaussian, and polynomial kernels. Optimise hyper-parameters with the inner-cross-validation procedure developed in assignment 3.b.

Report on the classification rate for the classification problems and RMSE for the regression problems. Present the comparison in a table, and report on each pair of results whether they are statistically significantly different using the TTEST2 function of Matlab (you can use ttest2 instead of ttest because you compare all results on the same testing data).

d. Additional Questions

Answer the following questions in your report:

Question 1: What does the kernel parameter of the Gaussian RBF kernel signify (sigma)? What happens when you increase its value?
Question 2: Explain what happens when a hard-margin SVM is fit to a dataset of two classes with overlapping features. What value do you need to set C (the slack-variable hyper-parameter) to attain a hard-margin SVM?

e. Deliverables

For the completion of Assignment 3, the following have to be handed in:

1. The code for training and evaluating SVMs, including the loading and transforming of the data, and code for comparing the three Machine Learning methods (ANN, DT, SVM) in a statistically principled manner.

2. A report of approximately two pages (not counting result matrices and graphs) containing: a cover sheet, a brief introduction, all the elements asked for in sections 9a-9d, and a brief conclusion.