Practical Workbook
CS-323/CS-412
Artificial Intelligence

Name : _____

Year : _____

Batch : _____

Roll No : _____

Department: _____

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology,

Practical Workbook
CS-323/CS-412
Artificial Intelligence

Prepared by:

Dr. Saad Qasim Khan

Revised in:

September 2019

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

INTRODUCTION

The Laboratory Workbook supports the Practical Sessions of the course Artificial Intelligence (CS-412). The Workbook has been designed to cover the major areas of Artificial Intelligence including Expert Systems, Machine Learning, Computer Vision and Fuzzy Logic Systems.

The Course Profile of CS-412 Artificial Intelligence lays down the following Course Learning Outcome:

"Practice system programming using a contemporary operating system (C3, PLO-3)"

All lab sessions of this workbook have been designed to assist the achievement of the above CLO. A rubric to evaluate student performance has been provided at the end of the workbook.

First part of this workbook is related to Machine Learning algorithms using Artificial Neural Networks (ANN), which is a problem solving paradigm, used to solve complex, non-linear problems where conventional algorithm solution is either not possible or not feasible. The section begins with laboratory session on implementation of basic logic function, and is followed by methods of creating and working on ANNs. Next lab session describes problems solving phases of ANNs; and finally the effect of external have been observed on the performance of ANNs.

The second part covers the basic and advanced concepts of developing Expert Systems. The two laboratory sessions discuss the syntax and usage of Specialization/Generalization definitions of Rules and Data Driven Programming. These two laboratory sessions covers the details of knowledge extraction and structures of a typical Expert System.

The third part explains how to build Fuzzy Logic based applications using Matlab Fuzzy Logic Toolbox. It also covers another tool Fuzzy Tech for building these applications

The Fourth section comprises on four laboratory sessions. The objective of the first laboratory session is to introduce the Machine learning tools (Matlab toolbox, Weka and PRAAT) for implementing various learning machines based applications. The remaining three laboratory sessions describe the principle and working methodology of the machine learning tools including exercise problems.

CONTENTS

Lab Session 01

Implementation of basic logic operations

Artificial Neural Network

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.
Artificial Neural Networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained Artificial Neural Network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer "what if" questions.
Other advantages include:

1. *Adaptive learning:* An ability to learn how to do tasks based on the data given for training or initial experience.
2. *Self-Organization:* An ANN can create its own organization or representation of the information it receives during learning time.
3. *Real Time Operation:* ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. *Fault Tolerance via Redundant Information Coding:* Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

A Simple Neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.

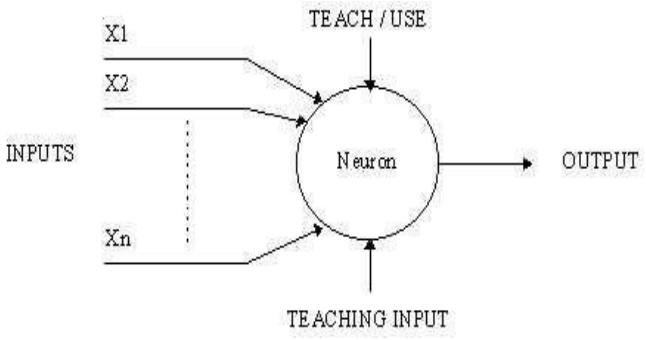


Figure 1.1: *A simple neuron*

Network layers

The commonest type of Artificial Neural Network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units, which is connected to a layer of **"output**" units. (See figure 1.2)

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behavior of the output units depends on the activity of the hidden units and the eights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

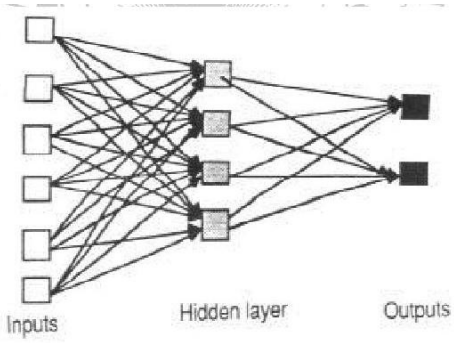


Figure 1.2: A Simple Feed Forward Network

We also distinguish single-layer and multi-layer architectures. The single-layer organization, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organizations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (See figure 1.3) turns out to be an MCP model (neuron with weighted inputs) with some additional, fixed, pre-processing. Units labeled A1, A2, Aj , Ap are called association units and their task is to extract specific,

Localized featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot more.

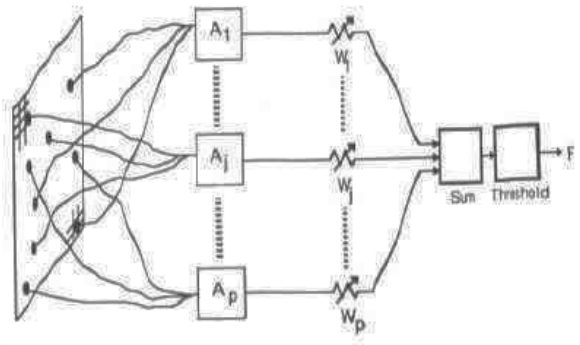


Figure 1.3: A Perceptron

Transfer Functions

The behavior of an ANN depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- For ***linear*** (or ramp) the output activity is proportional to the total weighted output.
- For ***threshold units***, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.
- For ***sigmoid units***, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurons than do linear or threshold units, but all three must be considered rough approximations.

To make an Artificial Neural Network that performs some specific task, we must choose how the units are connected to one another and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

We can teach a network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

Implementation of Logic Functions

In this practical we will learn how basic logic functions can be implemented and trained, using MS Excel. The procedure is explained by implementing a 2-input AND gate.
First of all you have to include following columns in your Excel sheet:

X_i : (column for inputs)
Z : (column for true output)
Y : (column for computed output)
D : (column for keeping track of the difference between true output & computed one)
W_i : (column for initial weights, assigned arbitrarily in the first step)
Wf : (column for final weights, which is computed from initial weight and becomes the initial weight of the first step)

Procedure

1. In input columns X_1 and X_2, include all possible values which can be provided to a 2-input AND gate, and in column *Z*, list all expected results.
2. In initial weights columns, W_1 & W_2, arbitrarily enter any values.
3. Apply following formula to Y (column for computed output);

$$Y = \Sigma\ W_i\ .\ X_i + bias$$

Where; *bias* is any constant (less than 1 for implementation of logic functions).

S	X_1	X_2	Z	W1i	W_{2i}	Y	D	W_{1f}	W_{2f}
1	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	1						
	1	0	1						
	1	1	1						

Figure 1.4: *A portion of the Excel Sheet*

4. Calculate the difference between true and computed outputs, using the formula

$$D = Z - Y$$

5. Calculate final weights by applying the following formula.

$$W_f = W_i + \alpha D X_i$$

Where ‗a' is the learning rate, which is arbitrarily assigned and preferably kept lesser than 0.5

6. Final weights computed in for first set of inputs are passed on as initial weights for second set of inputs, for the same iteration.
7. It is observed that after completing the first iteration, values of true and computed do not match for each possible set of inputs, i.e. difference is non-zero, so the process is repeated up to the point where this difference becomes zero.

Note: *If 0.1 is selected as initial weights and learning rate is kept 0.2, and bias is set to zero, then result is obtained in 4th iteration.*

Logic OR-Gate

Same method is followed for the implementation of logic OR-function.

Logic NOR & NAND Implementation

For the implementation of NOR and NAND gates, a positive bias is added to the weighted sum of inputs.

EXERCISES

1. Complete the following ables.

a.

Operation	Weights		Learning Rate	Iterations Required
	W_1	W_2		
OR	0.1	0.1	0.2	
	0.9	0.8	0.2	
	0.5	0.5	0.2	
	0.2	0.4	0.2	
	-0.3	-0.5	0.2	

b.

Operation	Weights		Learning Rate	Iterations Required
	W_1	W_2		
AND	0.9	0.9	0.2	
	0.1	0.1	0.2	
	0.5	0.5	0.2	
	0.9	0.7	0.2	
	-0.7	-0.8	0.4	

c.

Operation	Weights		Learning Rate	Bias	Iterations Required
	W_1	W_2			
NAND	0.3	0.4	0.2	0.9	
	0.9	0.9	0.2	0.7	
	0.8	0.6	0.2	0.9	
	0.9	0.7	0.2	0.2	
	-0.7	-0.8	0.4	0.8	

d.

Operation	Weights		Learning Rate	Bias	Iterations Required
	W_1	W_2			
NOR	0.2	0.4	0.2	0.8	
	0.8	0.9	0.2	0.8	
	0.8	0.9	0.2	0.9	
	0.8	0.9	0.2	0.6	
	0.3	0.4	0.4	0.6	

2. For NAND and NOR implementation, what is the effect of setting bias to value <0.5?

_____

3. Add a positive bias <0.3 to AND and OR gate‘s output and check how many iterations are required to get correct output.

_____

4. What are your observations, regarding the following?
   a. Lower value of learning rate is faster. (Yes/No)

   Reason: _____

   _____

   b. For OR gate implementation, smaller values of weights require less iterations for obtaining correct result. (Yes/No)

   Reason: _____

   _____

   c. For NAND gate implementation, larger values of weights require less iterations for obtaining correct result. (Yes/No)

   Reason: _____

   _____

5. Implement XOR and XNOR functions and give all the formula you used in the implementation. Draw the MLPs used for the implementation of above functions. Also mention the following:

6.

	XOR	***XNOR***
Learning rate		
Initial weights		
Bias		
Iterations required		

Attach the Excel Sheet here.

7. Implement 3-input AND, OR, NAND and NOR gates.

Attach Excel Sheets here.

Lab Session 02

Developing an Artificial Neural Network

EasyNN

EasyNN can be used to create, control, train, validate and query neural networks.

Getting Started

In order to create a neural network, press the **`New`** toolbar button or use the **`File>New`** menu command to produce a new neural network,
An empty **`Grid`** with a vertical line, a horizontal line and an underline marker will appear. The marker shows the position where a grid column and row will be produced. Press the **`enter`** key and you will be asked *"Create new Example row?"* - answer **`Yes`**.
You will then be asked *"Create new Input/Output column?"* - answer **`Yes`**.
You have now created a training example with one input. The example has no name and no value. Press the **`enter`** key again and you will open the **`Edit`** dialog (see figure 2.1). This dialog is used to enter or edit all of the information in the Grid.

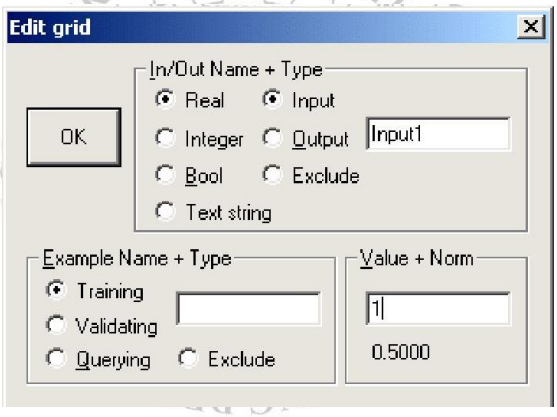


Figure 2.1: *Edit Dialog*

Enter value, in the **Value + Norm** edit box and then tab.
Type input name in the **Example Name + Type** edit box and then tab. The **Type** is already set to Training so just **tab** again.
By following above procedure, you can enter training data.
To create the neural network press the  toolbar button or use the **Action>New Network** menu command. This will open the _New Network'dialog. (See figure 2.2)

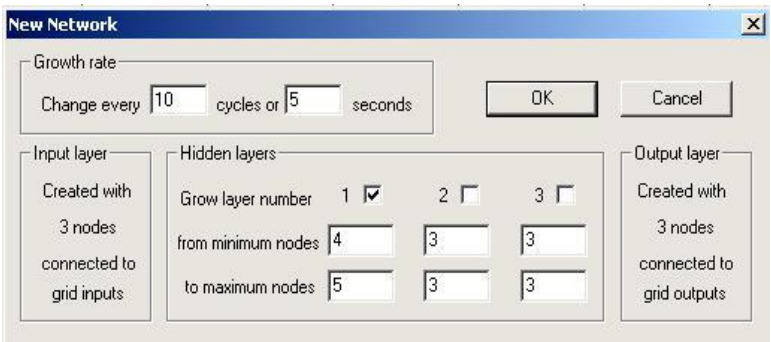


Figure 2.2: *New Network Dialog*

The neural network will be produced from the data you entered into the grid.

Press the  toolbar button or use the **`View>Network`** menu command to see the new neural network. This network will be somewhat like what is shown in figure 2.3

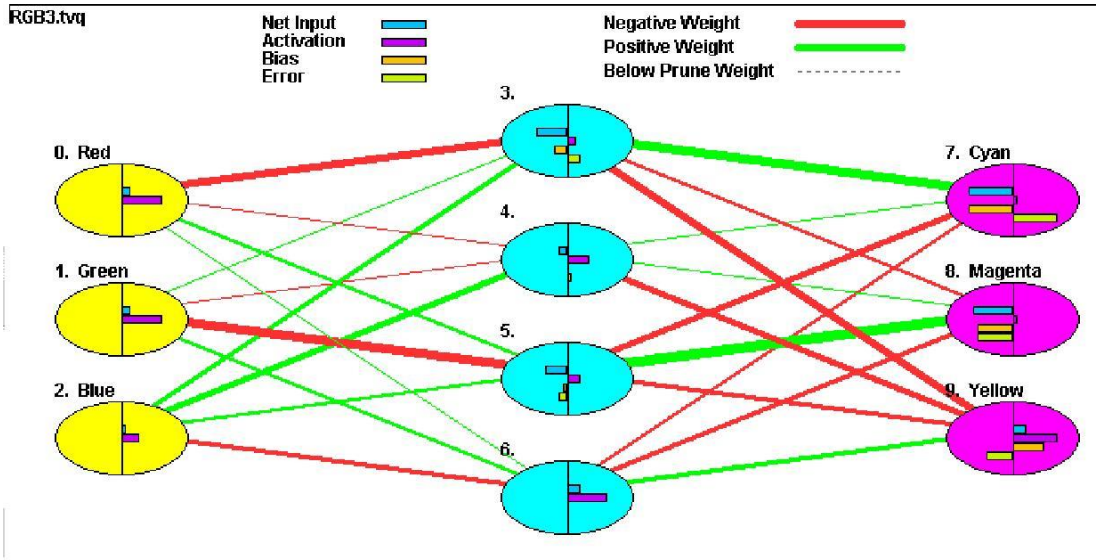


Figure 2.3: *Artificial Neural Network*

The neural network controls now need to be set. Press  toolbar button or sue **`Action>Change Controls`** menu command to open the **`Controls`** dialog. Check **`Optimize`** for both **`Learning Rate`** and **`Momentum`** and then press **`Ok`**. In this way, controls can be set and the neural network will be ready to learn the data that you entered into the grid. (See figure 2.4)

Press  toolbar button or use the **`Action>Start Learning`** menu command to open the **`Learning Progress`** dialog. The learning process will start and it will stop automatically, when the target error is reached. Press the **`Close`** button.

Press  toolbar button or use the **`View>Graph`** menu command to see how the error reduced to the target.

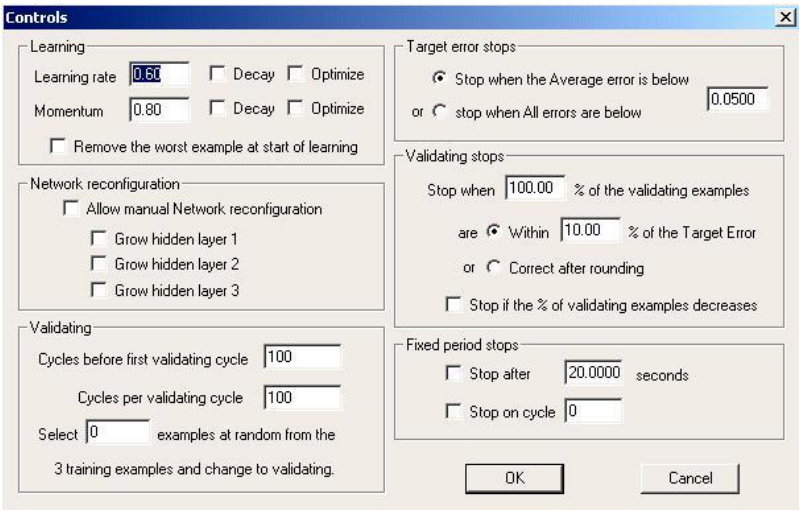


Figure 2.4: *Controls Dialog*

Press  toolbar button or use the **Action>Query** menu command to open the **Query** dialog. Press the **Add Query** button and the example named ―Query‖ will be generated and selected. Values for Query can be inserted here and output can be observed (See figure 2.5)

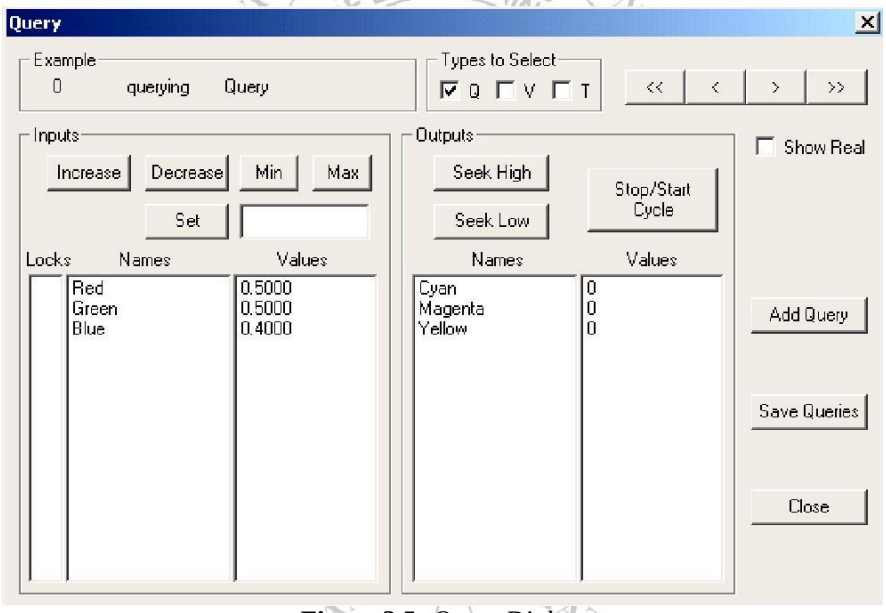


Figure 2.5: *Query Dialog*

Importing a File

Up till now, we have learned that how an artificial neural network can be grown and trained using EasyNN. Now we‘ll cover how real world data file, collected through different sources can be imported in EasyNN for neural network training.

Procedure

EasyNN can import a text file to create a new Grid or to add new example rows to an existing Grid in the following manner.

1. **File>New** to create a new Grid or **File>Open** to add rows to an existing Grid.
2. **File > Import** File
3. Open the file that is to be imported.

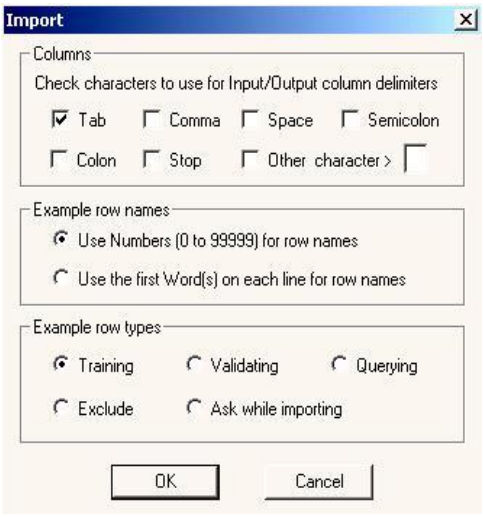


Figure 2.6

4. Check all the characters that are to be used for column delimiters.
5. Any words before the first delimiter on each line can be used for row names. If no row names are available then EasyNN can generate numbers for row names.
6. Press OK.

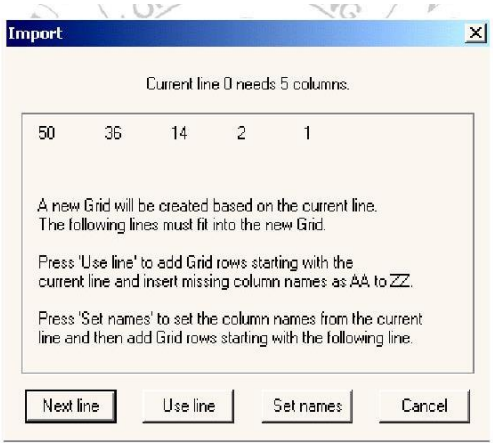


Figure 2.7

7. Press **`Next line`** until the first line to be imported is shown.
8. Press **`Use line`** or **`Set names`** according to the instructions in the dialog.

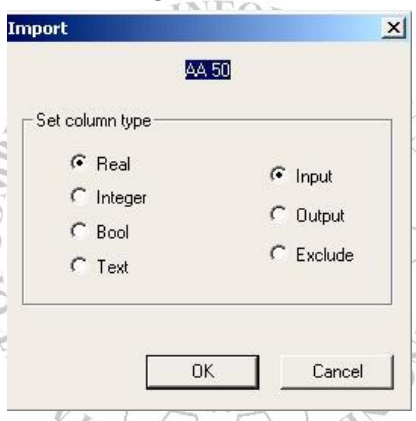


Figure 2.8

9. Set the column types when the first line is imported.
10. Press OK.
11. The rest of the file will be imported (See figure 2.9). Warnings will be produced for any lines in the file that are not suitable for the Grid.

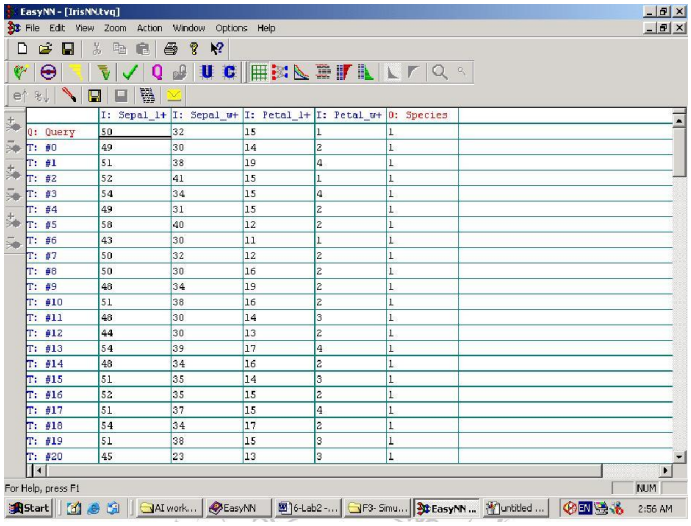


Figure 2.9

Note: *Network is grown and trained in the same manner as previously discussed.*

EXERCISES

1. Create a neural network, by using the training data, presented in the table. And test it on the given queries.

	Red	Green	Blue	Cyan	Magenta	Yellow	Output	Output	Type
	1	1	0	0	0	1	Y	0	Training
	1	0	1	0	1	0	M	0.5	Training
	0	1	1	1	0	0	C	1	Training
	0.9	0.9	0.3						Querying
	0.6	0.6	0.4						Querying
	0.5	0.5	0.5						Querying
	0	0	0						Querying
	1	1	1						Querying
	0.8	0.2	0.7						Querying

Learning rate selected: _____
Stop when average error is: _____

(a) Multiple output lines give better results. (Yes/ No)

Reason:

_____

_____

_____

2. Import a data file, provided by the instructor, grow an Artificial Neural Network, train it and test on different queries.

3. Give the following specification of your network.

(a) Learning rate selected : _____
(b) Stop when average error is: _____
(c) Inputs to the system and their type:

_____

_____

_____

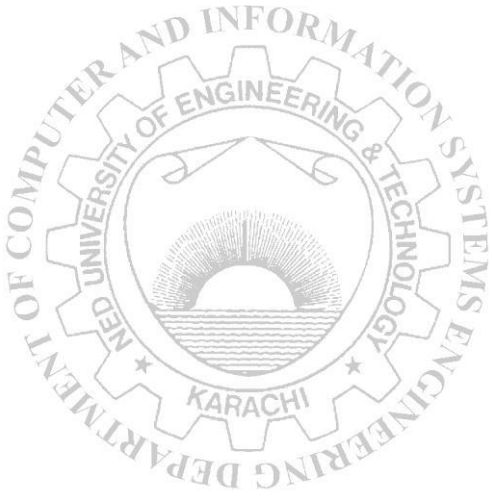
_____

_____

_____

_____

(d) Number of outputs and their types



Lab Session 03

Data Preprocessing for Artificial Neural Networks

There are a variety of parameters that play role in the success of any neural network solution. The External Parameters include the data. The quality, availability, reliability and relevance of the data used to develop and run the system are critical to its success. Even a primitive model can perform well if the input data has been processed in such a way that it clearly reveals the important information. On the other hand, even the best model cannot help us much if the necessary input information is presented in a complex and confusing way. Similarly, the internal parameters play a major role in the performance of ANN. These include the learning rate, momentum, number and size of layers etc.

Data-flow in a typical ANN system

The Data Flow sequence in a typical ANN-based system is shown in Figure 4.1.

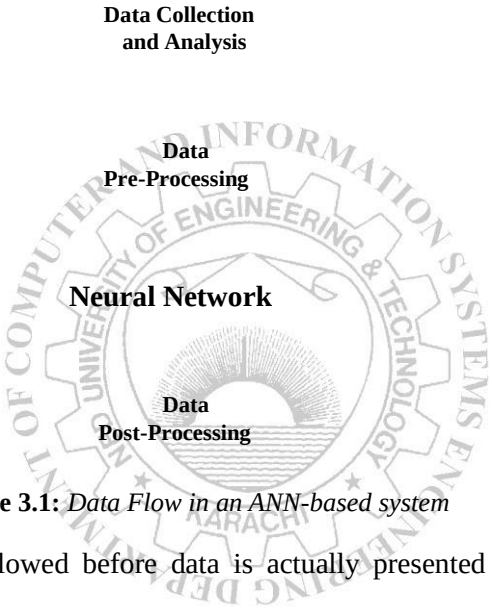


Figure 3.1: *Data Flow in an ANN-based system*

The following steps are to be followed before data is actually presented to the Artificial Neural Network:

I – Data Collection

The data collection plan typically consists of three tasks:

1 Identifying the data requirements

The first thing to do when planning data collection is to decide what data we will need to solve the problem. In general, it will be necessary to obtain the assistance of some experts in the field.

2 Identifying data sources

The next step is to decide from where the data will be obtained. This will allow us to make realistic estimates of the difficulty and expense of obtaining it. If the application demands real time data, these estimates should include an allowance for converting analogue data to digital form.

3 Determining the data quantity

It is important to make a reasonable estimation of how much data we will need to develop the neural network properly. If too little data is collected, it may not reflect the full range of properties that the network should be learning, and this will limit its performance with unseen data. On the other hand, it

is possible to introduce unnecessary expense by collecting too much data. In general, the quantity of data required is governed by the number of training cases that will be needed to ensure the network performs adequately.

II – Data Preparation

When the raw data has been collected, it may need converting into a more suitable format. At this stage, we should do the following:

1 Data validity checks

Data validity checks will reveal any patently unacceptable data that, if retained, would produce poor results. A simple data range check is an example of validity checking. For example, if we have collected oven temperature data in degrees centigrade, we would expect values in the range 50 ^{o}C to 400 ^{o}C. A value of, say, -10 ^{o}C, or 900 ^{o}C, is clearly wrong.

2 Partitioning data

Partitioning is the process of dividing the data into validation sets, training sets, and test sets. By definition, **validation sets** are used to decide the architecture of the network; **training sets** are used to actually update the weights in a network; **test sets** are used to examine the final performance of the network. The primary concerns should be to ensure that: a) the training set contains enough data, and suitable data distribution to adequately demonstrate the properties we wish the network to learn; b) there is no unwarranted similarity between data in different data sets.

III – Data Preprocessing

Data preprocessing consists of all the actions taken before the Neural Network comes into play. It is essentially a transformation T that transforms the raw real world data vectors X to a set of new data vectors Y:

$$Y = T(X)$$

such that:

i. Y preserves the ―valuable information‖ in X and
ii. Y is more useful than X.

IV – Training the ANN

This involves the following steps:

1. Transfer Data from Spreadsheet (E.g. MS - Excel) to ANN Tool (E.g. EasyNN)
2. Grow a Neural Network
3. Set Controls (optional)
4. Start Training
5. View Results

V – Querying the ANN

Querying the Network involves the following steps:

1. Transfer data from Spreadsheet to Tool
2. Query the Network
3. View Results

Practical ANN-based Systems

Artificial Neural Networks are being used in various application domains, like prediction, classification, diagnosis and forecasting etc in different situations like Stock Rate prediction, weather forecasting, Pattern Recognition and Medical Applications etc.

An ANN based Heart Disease Diagnosis System

The case study has been taken up to diagnose the presence or absence of Heart Disease in clients. As mentioned above, there are certain steps that are to be considered before ANN comes into play. As mentioned above, these steps include:

Data Collection – A database of 75 attributes has been constructed for HDDS. After thorough analysis, irrelevant and redundant parameters have been removed. The final database comprises of 13 parameters, on the basis of which, the output is computed. These are:

i. Age (Real Number)
ii. Gender (Binary)
iii. Chest pain type (4 values) (Real Number)
iv. Resting blood pressure (Real Number)
v. Serum cholesterol in mg/dl (Real Number)
vi. Fasting blood sugar > 120 mg/dl (Binary)
vii. Resting electrocardiographic results (values 0,1,2) (Real Number)
viii. Maximum heart rate achieved (Real Number)
ix. Exercise induced angina (Binary)
x. Old peak = ST depression induced by exercise relative to rest (Real Number)
xi. The slope of the peak exercise ST segment (Real Number)
xii. Number of major vessels (0-3) colored by fluoroscopy (Ordered)
xiii. Thal: 3 = normal; 6 = fixed defect; 7 = reversible defect (Real Number)

The output of the system is either absence or presence of Heart Disease (represented by 1 and 2 respectively). A total of 270 cases have been considered. Out of these cases, 55.56% showed an absence of disease and 44.44% showed presence of disease. It has been ensured that the dataset contains nearly equal percentage of both the classes of output.

A portion of the dataset is given here:

44	0	3	108	141	0	0	175	0	0.6	2	0	3	1
71	0	4	112	149	0	0	125	0	1.6	2	0	3	1
45	0	2	112	160	0	0	138	0	0	2	0	3	1
57	1	3	150	168	0	0	174	0	1.6	1	0	3	1
65	1	4	120	177	0	0	140	0	0.4	1	0	7	1
46	0	3	142	177	0	2	160	1	1.4	3	0	3	1

77	1	4	125	304	0	2	162	1	0	1	3	3	2
56	0	4	200	288	1	2	133	1	4	3	2	7	2
67	1	4	120	237	0	0	71	0	1	2	0	3	2
54	1	2	192	283	0	2	195	0	0	1	1	7	2
62	0	4	160	164	0	2	145	0	6.2	3	3	7	2
67	1	4	160	286	0	2	108	1	1.5	2	3	3	2

There are some companies that collect and provide such statistical data, which may be used in Neural network based applications.

Data Preparation

This step involves Validity checking and Partitioning of Data. The data has been partitioned into two parts – The training dataset and the test dataset. It has been ensured that the minimum and maximum value of each parameter lies in the training dataset because the ANN generates a working range for each parameter and values outside this range are clipped off.

Data Preprocessing

The data has been preprocessed using the scaling technique, and then used to train the ANN. The sample scaled data is shown here:

0.3125	0	0.666667	0.13207	0.03424	0	0	0.793893	0	0.09677	0.5	0	0	1
0.875	0	1	0.16981	0.05251	0	0	0.412213	0	0.25806	0.5	0	0	1
0.33333	0	0.333333	0.16981	0.07762	0	0	0.511450	0	0	0.5	0	0	1
0.58333	1	0.666667	0.52830	0.09589	0	0	0.786259	0	0.25806	0	0	0	1
0.75	1	1	0.24528	0.11643	0	0	0.526717	0	0.06451	0	0	1	1
0.35416	0	0.666667	0.45283	0.11643	0	1	0.679389	1	0.22580	1	0	0	1
0.5625	0	1	0.37735	0.64611	0	1	0.603053	1	0.30645	0.5	0.6667	1	2
1	1	1	0.29245	0.40639	0	1	0.694656	1	0	0	1	0	2
0.5625	0	1	1	0.36986	1	1	0.473282	1	0.64516	1	0.6667	1	2
0.79166	1	1	0.24528	0.25342	0	0	0	0	0.16129	0.5	0	0	2
0.52083	1	0.333333	0.92452	0.35844	0	1	0.946564	0	0	0	0.3333	1	2
0.6875	0	1	0.62264	0.08675	0	1	0.564885	0	1	1	1	1	2

Training the ANN

1. Training involves importing the _train dataset‘ for HDDS in the tool.
2. The Neural Network is then generated and various parameters are defined, along with the terminating criterion.
3. The training begins once the controls are set and is terminated once the criterion is met.

Testing the ANN

Once training is complete, the HDDS is tested for authenticity. This may be accomplished by either inserting the queries manually or by importing the _HDDS test dataset‘ into the tool.

On the basis of training, the ANN based HDDS predicts the presence or absence of Heart Disease in patients.

EXERCISES

1. Come up with some Real-world problem, generate and train an ANN for it and then query the network. Mention the Input and Output parameters.

_____

_____

_____

2. In continuation with Exercise 1 fill in the following entries:

Number of Output Categories: _____

Number of Data Rows: _____

Number of Training Rows: _____

Number of Testing Rows: _____

Learning Rate: _____

Momentum: _____

Number of Layers: _____

Size of Layers:

_____

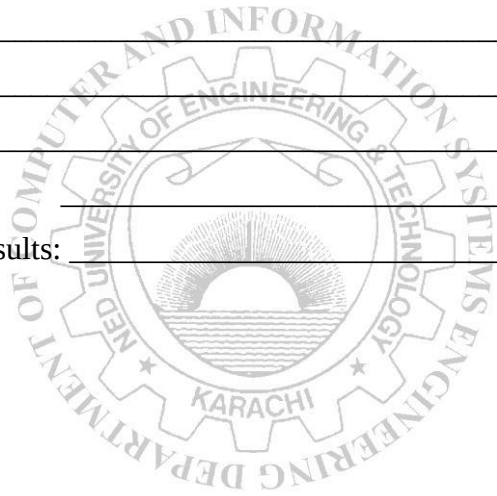
_____

_____

_____

Number of Cycles for Training: _____

Percentage of Correctness in Results: _____



Lab Session 04

OBJECT

Learning Algorithmic Design of Artificial Neural Network

This lab session will introduce the concept of Algorithmic design of Artificial Neural Network (ANN). The algorithmic design of ANNs is necessary to understand the design philosophy of ANN implementations on software programming languages.

MATLAB, an abbreviation for 'matrix laboratory,' is a platform for solving mathematical and scientific problems. It is a proprietary programming language developed by MathWorks, allowing matrix manipulations, functions and data plotting, algorithm implementation, user interface creation and interfacing with programs written in programming languages like C, C++, Java and so on.

Widrow and Hoff [1960] development developed the learning rule that is very closely related to the perceptron learning rule. The rule, called Delta rule, adjusts the weights to reduce the difference between the net input to the output unit, and the desired output, which results in the least mean squared (LMS error). Adaline (Adaptive Linear Neuron) and Madaline (Multilayered Adaline) networks use this LMS learning rule and are applied to various neural network applications.

Adaline is found to use bipolar activations for its input signals and target output. The weights and the bias of the Adaline are adjustable. The learning rule used can be called as Delta rule, Least Mean Square rule or Widrow-Hoff rule. The derivation of this rule with single output unit, several output units. Since the activation function is an identity function, the activation of the unit is its net input.

When Adaline is to be used for pattern classification, then, after training, a threshold function is applied to the net input to obtain the activation. The Adaline unit can solve the problem with linear separability if it occurs.

Architecture

The architecture of an adaline is shown in Fig. 4.1.

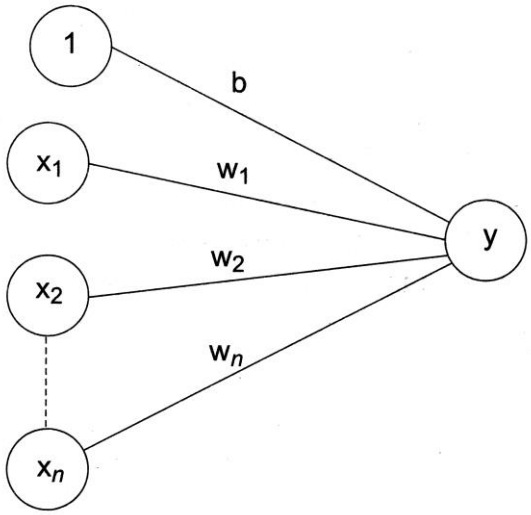


Figure 4.1: Architecture of Adaline Network

The Adaline has only one output unit. This output unit receives input from several units and also from bias; whose activation is always +1. The adaline also resembles a single layer network. It receives

input from several neurons. It should be noted that it also receives input from the unit which is always '+1' called as bias. The bias weights are also trained in the same manner as the other weights. In Fig. 1, an input layer with $x_1 \ldots x_i \ldots x_n$ and bias, an output layer with only one output neuron is present. The link between the input and output neurons possess weighted interconnections. These weights get changed as the training progresses.

Algorithm

Basically, the initial weights of Adaline network have to be set to small random values and not to zero as discussed in Hebb or perceptron networks, because this may influence the error factor to be considered. After the initial weights are assumed, the activations for the input unit are set. The net input is calculated based on the training input patterns and the weights. By applying delta learning rule discussed in 3.3.3, the weight updation is being carried out. The training process is continued until the error, which is the difference between the target and the net input becomes minimum. The step based training algorithm for an adaline is as follows:

Pseudo Code for Adaline network

Training Algorithm

The following is the pseudo code for training an Adaline network.

Step 1: Initialize weights (not zero but small random values are used). Set learning rate α.

Step 2: While stopping condition is false, do Step 3-7.

Step 3: For each bipolar training pair s: t, perform Steps 4-6.

Step 4: Set activations of input units $x_i = s_i$ for i 1 to n.

Step 5: Compute net input to output unit $y_{-in} = b + \Sigma x_i w_i$

Step 6: Update bias and weights, i = 1 to n.

$$w_i(new) = w_i(old) + a(t - y_{-in})x_i$$

$$b(new) = b(old) + \alpha(t - y_{-in})$$

Step 7: Test for stopping condition.

The stopping condition may be when the weight change reaches small level or number of iterations etc.

Testing Algorithm

The application procedure, which is used for testing the trained network is as follows. It is mainly based on the bipolar activation.

Step 1: Initialize weights obtained from the training algorithm.

Step 2: For each bipolar input vector x, perform Steps 3-5.

Step 3: Set activations of input unit.

Step 4: Calculate the net input to the output unit. $y_{-in} = b + \Sigma x_i w_i$

Step 5: Finally apply the activations to obtain the output y.

$$y = f(y - in) = \begin{cases} 1, & if\ y - in \geq 0 \\ -1, & if\ y - in < 0 \end{cases}$$

EXERCISE:

Develop a MATLAB program for OR function with bipolar inputs and targets using Adaline network. The truth table for the OR function with bipolar inputs and targets is given as,

X1	X2	Y
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

Lab Session 05

Python Based Implementation of Artificial Neural Network

Designing Artificial Neural Network in Python

Python is a high-level programming language which is applicable in every computing domain. It was developed in 1991 by Guido van Rossum. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python offers significant code readability with easy to program language constructs.

Python offers extensive library support for designing Artificial Neural Network (ANN). In Python, ANN algorithm can be written in simple syntax or frameworks may be used for ANN implementation. Python offers a large number of frameworks for implementation of ANNs from a simple perceptron level implementation to deep neural networks such as Convolutional Neural Networks (CNNs) and Long-Short Term Memory Networks (LSTM).

In Python programming environment, numpy library is used for scientific and linear algebra operations. Supposing that Python and pip are already installed on your system, numpy can be installed by running the following command on the python command prompt.

```
pip install numpy
```

In this lab, a very simple network with 2 layers is selected for implementation. In order to do that a very simple dataset is required, so the XOR dataset in used for implementation. The Truth table for XOR gate along with network diagram is shown in Fig. 5.1 and 5.2, respectively.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5.1: Truth Table for XOR gate

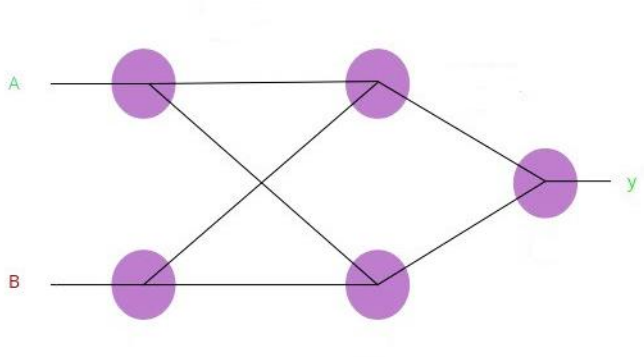


Figure 5.2: Network diagram for XOR gate

Implementation in Python

Here, the parameters to be learned are the weights W1, W2 and the biases b1, b2. From the basic theory of NNs we know that the activations A1 and A2 are calculated as following:

```
A1 = h(W1*X + b1)
A2 = g(W2*A1 + b2)
```

Where g and h are the two activation functions we chose(for us sigmoid and tanh) and W1, W1, b1, b2 are generally Matrices.

First we will implement our sigmoid activation function defined as follows: g(z) = 1/(1+e^(-z)) where z will be a matrix in general. Luckily numpy supports calculations with matrices so the code is relatively simple:

```
import numpy as np
     def sigmoid(z):
       return 1/(1 + np.exp(-z))
```

Next we have to initialize our parameters. Weight matrices W1 and W2 will be randomly initialized from a normal distribution while biases b1 and b2 will be initialized to zero. The function initialize_parameters(n_x, n_h, n_y) takes as input the number of units in each of the 3 layers and initializes the parameters properly:

```
def initialize_parameters(n_x, n_h, n_y):
  W1 = np.random.randn(n_h, n_x)
  b1 = np.zeros((n_h, 1))
  W2 = np.random.randn(n_y, n_h)
  b2 = np.zeros((n_y, 1))

parameters
= {
          "W1": W1,
          "b1" : b1,
          "W2": W2,
          "b2" : b2
          }
          return parameters
```

The next step is to implement the Forward Propagation. The function forward_prop(X, parameters) takes as input the neural network input matrix X and the parameters dictionary and returns the output of the NN A2 with a cache dictionary that will be used later in back propagation.

```
def
forward_prop(X,
parameters):
            W1 = parameters["W1"]
            b1 = parameters["b1"]
```

```
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache = {
        "A1": A1,
        "A2": A2
    }
    return A2, cache
```

We now have to compute the loss function. We will use the Cross Entropy Loss function. Calculate_cost(A2, Y) takes as input the result of the NN A2 and the ground truth matrix Y and returns the cross entropy cost:

```
def calculate_cost(A2, Y):
  cost = -np.sum(np.multiply(Y, np.log(A2)) +  np.multiply(1-Y,
np.log(1-A2)))/m
  cost = np.squeeze(cost)
  return cost
```

Now the most difficult part of the Neural Network algorithm, Back Propagation. The code here may seem a bit weird and difficult to understand but we will not dive into details of why it works here. This function will return the gradients of the Loss function with respect to the 4 parameters of our network (W1, W2, b1, b2):

```
def backward_prop(X, Y, cache, parameters):
  A1 = cache["A1"]
  A2 = cache["A2"]
  W2 = parameters["W2"]
  dZ2 = A2 - Y
  dW2 = np.dot(dZ2, A1.T)/m
  db2 = np.sum(dZ2, axis=1, keepdims=True)/m
  dZ1 = np.multiply(np.dot(W2.T, dZ2), 1-np.power(A1, 2))
  dW1 = np.dot(dZ1, X.T)/m
  db1 = np.sum(dZ1, axis=1, keepdims=True)/m
  grads = {
    "dW1": dW1,
    "db1": db1,
    "dW2": dW2,
    "db2": db2
  }
  return grads
```

We will use **Gradient Descent** algorithm to update our parameters and make our model learn with the

learning rate passed as a parameter:

```
def update_parameters(parameters, grads, learning_rate):
  W1 = parameters["W1"]
  b1 = parameters["b1"]
  W2 = parameters["W2"]
  b2 = parameters["b2"]
  dW1 = grads["dW1"]
  db1 = grads["db1"]
  dW2 = grads["dW2"]
  db2 = grads["db2"]
  W1 = W1 - learning_rate*dW1
  b1 = b1 - learning_rate*db1
  W2 = W2 - learning_rate*dW2
  b2 = b2 - learning_rate*db2

  new_parameters = {
    "W1": W1,
    "W2": W2,
    "b1" : b1,
    "b2" : b2
  }
  return new_parameters
```

By now we have implemented all the functions needed for one circle of training. Now all we have to do is just put them all together inside a function called model() and call model() from the main program.
Model() function takes as input the features matrix X, the labels matrix Y, the number of units n_x, n_h, n_y, the number of iterations we want our Gradient Descent algorithm to run and the learning rate of Gradient Descent and combines all the functions above to return the trained parameters of our model:

```
def model(X, Y, n_x, n_h, n_y, num_of_iters, learning_rate):
parameters = initialize_parameters(n_x, n_h, n_y)
  for i in range(0, num_of_iters+1):
    a2, cache = forward_prop(X, parameters)
    cost = calculate_cost(a2, Y)
    grads = backward_prop(X, Y, cache, parameters)
    parameters = update_parameters(parameters, grads,
learning_rate)
    if(i%100 == 0):
      print('Cost after iteration# {:d}: {:f}'.format(i, cost))
  return parameters
```

The **training part** is now over. The function above will return the trained parameters of our NN. Now we just have to make our **prediction**. The function predict(X, parameters) takes as input the matrix X with elements the 2 numbers for which we want to compute the XOR function and the trained parameters of the model and returns the desired result y_predict by using a threshold of 0.5:

```
def
predict(X,
parameters):
              a2, cache = forward_prop(X, parameters)
              yhat = a2
              yhat = np.squeeze(yhat)
              if(yhat >= 0.5):
                y_predict = 1
              else:
                y_predict = 0
              return y_predict
```

We are done with all the functions needed finally. Now let's go to the main program and declare our matrices X, Y and the hyperparameters n_x, n_h, n_y, num_of_iters, learning_rate:

```
np.random.seed(2)
                # The 4 training examples by columns
                X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
                # The outputs of the XOR for every example in X
                Y = np.array([[0, 1, 1, 0]])
                # No. of training examples
                m = X.shape[1]
                # Set the hyperparameters
                n_x = 2     #No. of neurons in first layer
                n_h = 2     #No. of neurons in hidden layer
                n_y = 1     #No. of neurons in output layer
                num_of_iters = 1000
                learning_rate = 0.3
```

Having all the above set up, training the model on them is as easy as calling this line of code:

```
    trained_parameters = model(X, Y, n_x, n_h, n_y, num_of_iters,
                               learning_rate)
```

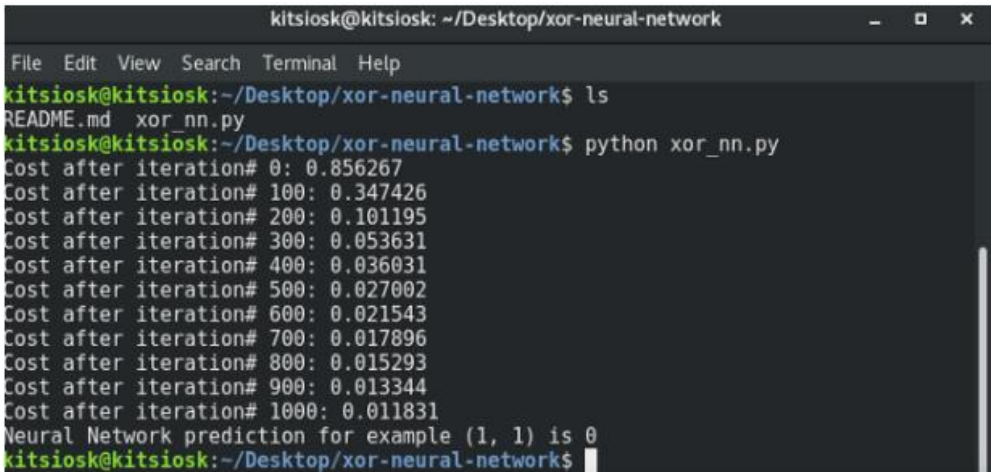
Finally let's make our prediction for a random pair of numbers, let's say (1,1):

```
# Test 2X1 vector to calculate the XOR of its elements.
# You can try any of those: (0, 0), (0, 1), (1, 0), (1, 1)
X_test = np.array([[1], [1]])
y_predict = predict(X_test, trained_parameters)
# Print the result
print('Neural Network prediction for example ({:d}, {:d}) is
{:d}'.format(
    X_test[0][0], X_test[1][0], y_predict))
```

That was the actual code! Let's see our results. If we run our file, let's say xor_nn.py, with this command

```
python xor_nn.py
```

we get the following result, that is indeed correct because 1XOR1=0.



Implementation of Logic Functions

Complete Code

This code is written by Konstantinos Kitsios. Available at https://gitlab.com/kitsiosk/xor-neural-net/blob/master/xor_nn.py.

```
"""
    Simple Neural Network with 1 hidden layer with the number
    of hidden units as a hyperparameter to calculate the XOR
function
"""

import numpy as np

def sigmoid(z):
    return 1/(1 + np.exp(-z))

def initialize_parameters(n_x, n_h, n_y):
    W1 = np.random.randn(n_h, n_x)
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)
    b2 = np.zeros((n_y, 1))

    parameters = {
        "W1": W1,
        "b1" : b1,
        "W2": W2,
        "b2" : b2
    }
    return parameters

def forward_prop(X, parameters):
```

```
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = {
        "A1": A1,
        "A2": A2
    }
    return A2, cache

def calculate_cost(A2, Y):
    cost = -np.sum(np.multiply(Y, np.log(A2)) +  np.multiply(1-Y,
np.log(1-A2)))/m
    cost = np.squeeze(cost)

    return cost

def backward_prop(X, Y, cache, parameters):
    A1 = cache["A1"]
    A2 = cache["A2"]

    W2 = parameters["W2"]

    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T)/m
    db2 = np.sum(dZ2, axis=1, keepdims=True)/m
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1-np.power(A1, 2))
    dW1 = np.dot(dZ1, X.T)/m
    db1 = np.sum(dZ1, axis=1, keepdims=True)/m

    grads = {
        "dW1": dW1,
        "db1": db1,
        "dW2": dW2,
        "db2": db2
    }

    return grads

def update_parameters(parameters, grads, learning_rate):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]
```

```
    W1 = W1 - learning_rate*dW1
    b1 = b1 - learning_rate*db1
    W2 = W2 - learning_rate*dW2
    b2 = b2 - learning_rate*db2

    new_parameters = {
        "W1": W1,
        "W2": W2,
        "b1" : b1,
        "b2" : b2
    }

    return new_parameters


def model(X, Y, n_x, n_h, n_y, num_of_iters, learning_rate):
    parameters = initialize_parameters(n_x, n_h, n_y)

    for i in range(0, num_of_iters+1):
        a2, cache = forward_prop(X, parameters)

        cost = calculate_cost(a2, Y)

        grads = backward_prop(X, Y, cache, parameters)

        parameters = update_parameters(parameters, grads,
learning_rate)

        if(i%100 == 0):
            print('Cost after iteration# {:d}: {:f}'.format(i,
cost))

    return parameters

def predict(X, parameters):
    a2, cache = forward_prop(X, parameters)
    yhat = a2
    yhat = np.squeeze(yhat)
    if(yhat >= 0.5):
        y_predict = 1
    else:
        y_predict = 0

    return y_predict

np.random.seed(2)

# The 4 training examples by columns
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])

# The outputs of the XOR for every example in X
Y = np.array([[0, 1, 1, 0]])

# No. of training examples
m = X.shape[1]
```

```
# Set the hyperparameters
n_x = 2     #No. of neurons in first layer
n_h = 2     #No. of neurons in hidden layer
n_y = 1     #No. of neurons in output layer
num_of_iters = 1000
learning_rate = 0.3

trained_parameters = model(X, Y, n_x, n_h, n_y, num_of_iters,
learning_rate)

# Test 2X1 vector to calculate the XOR of its elements.
# Try (0, 0), (0, 1), (1, 0), (1, 1)
X_test = np.array([[1], [1]])

y_predict = predict(X_test, trained_parameters)

print('Neural Network prediction for example ({:d}, {:d}) is
{:d}'.format(
    X_test[0][0], X_test[1][0], y_predict))
```

EXERCISES

Use the above lab example for XNOR gate implementation and retrain the network. Write down the modifications (only) to the above mentioned Python code for XNOR implementation.

Lab Session 6

Introduction to Image Processing on MATLAB

MATLAB can be used to perform wide variety of digital image processing operations such as image segmentation, image enhancement, noise reduction, geometric transformations, image registration and 3D image processing operations. Many of the functions support C/C++ code generation for desktop prototyping and embedded vision system deployment.

What is a digital image?

A digital image may be defined as a two-dimensional function f(x,y), where 'x' and 'y' are spatial coordinates and the amplitude of 'f' at any pair of coordinates is called the intensity of the image at that point. When 'x,' 'y' and amplitude values of 'f' are all finite discrete quantities, the image is referred to as a digital image.
Digitising the coordinate values is referred to as 'sampling,' while digitising the amplitude values is called 'quantisation.' The result of sampling and quantisation is a matrix of real numbers.

A digitised image is represented as:

$$f(x, y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \vdots & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix}$$

Each element in the array is referred to as a pixel or an image element.

Reading images

Images are read in MATLAB environment using the function 'imread.' Syntax of imread is:

```
imread('filename');
```

where 'filename' is a string having the complete name of the image, including its extension.
For example,

```
>>F = imread(Penguins_grey.jpg);
 >>G = imread(Penguins_RGB.jpg);
```

Please note that when no path information is included in 'filename,' 'imread' reads the file from the current directory. When an image from another directory has to be read, the path of the image has to be specified.

Image display

Images are displayed on the MATLAB desktop using the function imshow, which has the basic

syntax:

```
imshow(f)
```

Where 'f' is an image array of data type 'uint8' or double. Data type 'uint8' restricts the values of integers between 0 and 255.

Multiple images can be displayed within one figure using the subplot function. This function has three parameters within the brackets, where the first two parameters specify the number of rows and columns to divide the figure. The third parameter specifies which subdivision to use. For example, subplot(3,2,3) tells MATLAB to divide the figure into three rows and two columns and set the third cell as active. To display the images Penguins_RGB.jpg and Penguins_grey.jpg together in a single figure, you need to give the following commands:

```
>> A=imread('Penguins_grey.jpg');
>> B=imread('Penguins_RGB.jpg');
>>figure
>>subplot(1,2,1),imshow(A)
>>subplot(1,2,2),imshow(B)
```

The Image tool in the image processing toolbox provides a more interactive environment for viewing and navigating within images, displaying detailed information about pixel values, measuring distances and other useful operations. To start the image tool, use the imtool function. The following statements read the image Penguins_grey.jpg saved on the desktop and then display it using 'imtool':

```
>>B = imread(Penguins_grey.jpg);

>>imtool(B)
```

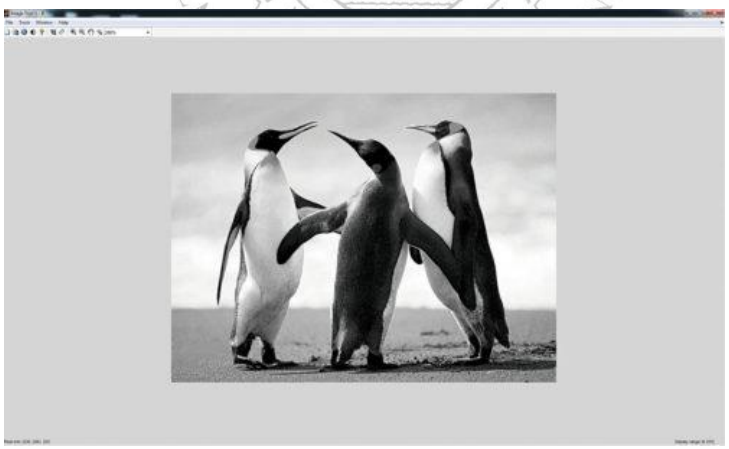


Figure 6.1 The image tool window with Penguins image

The Measure Distance tool under Tools tab is used to show the distance between the two selected points. Fig. 2 shows the use of this tool to measure distances between the beaks of different penguins.

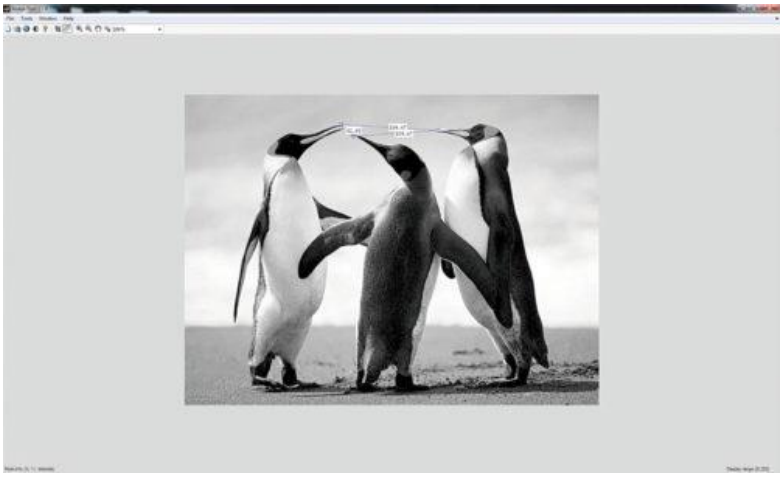


Figure 6.2 The Measure Distance tool under Tools tab is used to measure distances

Writing images

Images are written to the current directory using the imwrite function, which has the following syntax:

```
imwrite(f, 'filename');
```

This command writes image data 'f' to the file specified by 'filename' in your current folder. The imwrite function supports most of the popular graphic file formats including GIF, HDF, JPEG or JPG, PBM, BMP, PGM, PNG, PNM, PPM and TIFF and so on.
The following example writes a 100×100 array of grayscale values to a PNG file named random.png in the current folder:

```
>> F = rand(100);
>>imwrite(F, 'random.png')
```

For JPEG images, the imwrite syntax applicable is:

```
imwrite(f, 'filename.jpg', 'quality', q)
```

Where 'q' is an integer between 0 and 100.

It can be used to reduce the image size. The quality parameter is used as a trade-off between the resulting image's subjective quality and file size.
You can now read the image Penguins_grey.jpg saved in the current working folder and save it in JPG format with three different quality parameters: 75 (default), 10 (poor quality and small size) and 90 (high quality and large size):

```
F = imread('Penguins_grey.jpg');
imwrite(F,'Penguins_grey_75.
jpg','quality',75);
imwrite(F,'Penguins_grey_10.
jpg','quality',10);
imwrite(F,'Penguins_grey_90.
jpg','quality',90);
```

Figs 3.a and 3.b show images for quality factors of 10 and 90, respectively. You will note that that a low-quality image has a smaller size than a higher-quality image.

Figure 3.a

Figure 3.b

Figure 6.3: (a) Image with quality=10 and (b) Image with quality= 90

Image processing covers a wide and diverse array of techniques and algorithms. Fundamental processes underlying these techniques include sharpening, noise removal, deblurring, edge extraction, binarisation, contrast enhancement, and object segmentation and labeling.
Sharpening enhances the edges and fine details of an image for viewing by human beings. It increases the contrast between bright and dark regions to bring out image features. Basically, sharpening involves application of a high-pass filter to an image.

The command used for sharpening an image in MATLAB is:

```
B = imsharpen(A)
```

Noise removal techniques reduce the amount of noise in an image before it is processed any further. It is necessary in image processing and image interpretation so as to acquire useful information. Images from both digital cameras as well as conventional film cameras pick up noise from a variety of sources. These noise sources include salt-and-pepper noise (sparse light and dark disturbances) and Gaussian noise (each pixel value in the image changes by a small amount). In either case, the noise at different pixels can either be correlated or uncorrelated. In many cases, noise values at different pixels are modeled as being independent and identically distributed and hence uncorrelated. In selecting noise reduction algorithm, one must consider the available computer power and time and whether sacrificing some image detail is acceptable if it allows more noise to be removed and so on.
Deblurring is the process of removing blurring artifacts (such as blur caused by defocus aberration or motion blur) from images. The blur is typically modelled as a convolution point-spread function with a hypothetical sharp input image, where both the sharp input image (which is to be recovered) and the point-spread function are unknown. Deblurring algorithms include methodology to remove the blur from an image. Deblurring is an iterative process and you might need to repeat the process multiple times until the final image is the best approximation of the original image.

Read image into the workspace and display it.

```
I = imread('eight.tif');
figure
imshow(I)
```

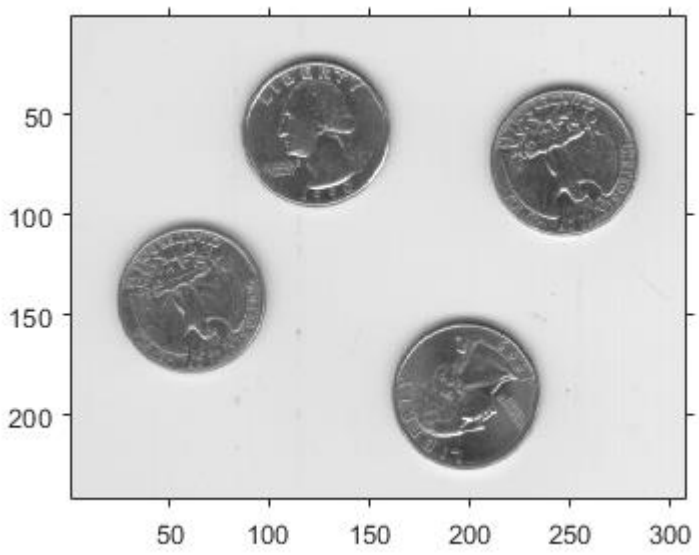


Figure 6.4: Original Image

For this example, add salt and pepper noise to the image. This type of noise consists of random pixels being set to black or white (the extremes of the data range).

```
J = imnoise(I,'salt & pepper',0.02);
figure
imshow(J)
```

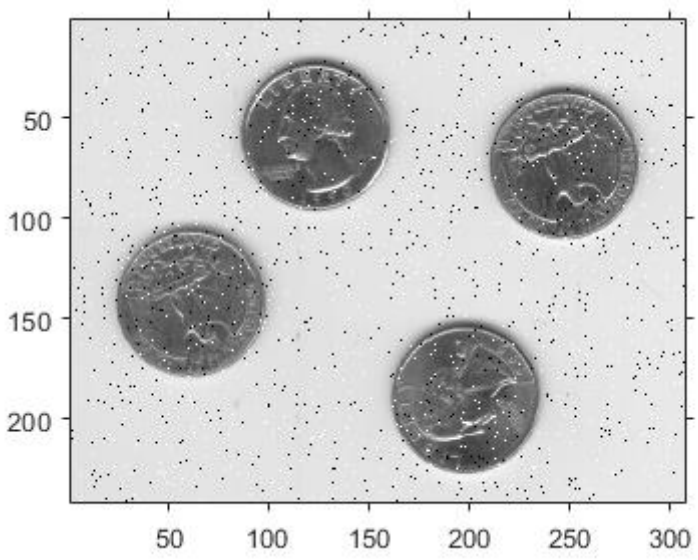


Figure 6.5: Image with salt and paper noise

Filter the noisy image, J, with an averaging filter and display the results. The example uses a 3-by-3 neighborhood.

```
Kaverage = filter2(fspecial('average',3),J)/255;
figure
imshow(Kaverage)
```

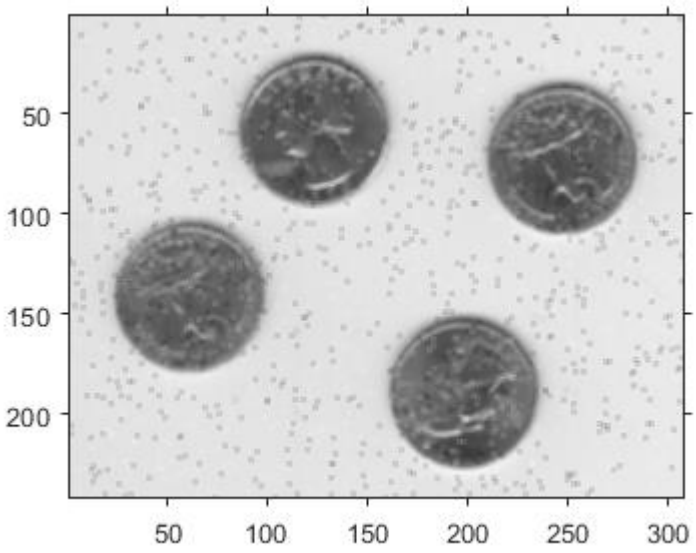


Figure 6.6: Image after Filtering

Now use a median filter to filter the noisy image, J. The example also uses a 3-by-3 neighborhood. Display the two filtered images side-by-side for comparison. Notice that medfilt2 does a better job of removing noise, with less blurring of edges of the coins.

```
Kmedian = medfilt2(J);
imshowpair(Kaverage,Kmedian,'montage')
```

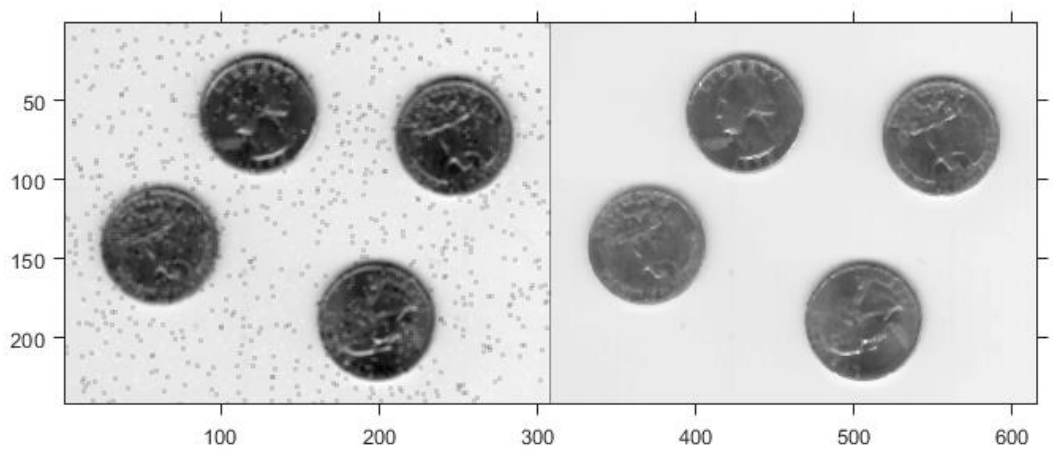


Figure 6.7: Image after Average and Median Filtering

Edge extraction or edge detection is used to separate objects from one another before identifying their contents. It includes a variety of mathematical methods that aim at identifying points in a digital image at which the image brightness changes sharply.

Edge detection approaches can be categorised into search-based approach and zero-crossing-based approach. Search based methods detect edges by first computing a measure of edge strength (usually a first-order derivative function) such as gradient magnitude and then searching for a local directional maxima of the gradient magnitude using a computed estimate of the local orientation of the edge, usually the gradient direction. Zero-crossing methods look for zero-crossings in a second-order derivative function computed from the image to find the edge. First-order edge detectors include Canny edge detector, Prewitt and Sobel operators, and so on.

Other approaches include second-order differential approach of detecting zero-crossings, phase congruency (or phase coherence) methods or phase-stretch transform (PST). Second-order differential approach detects zero-crossings of the second-order directional derivative in the gradient direction. Phase congruency methods attempt to find locations in an image where all sinusoids in the frequency domain are in phase. PST transforms the image by emulating propagation through a diffractive medium with engineered 3D dispersive property (refractive index).

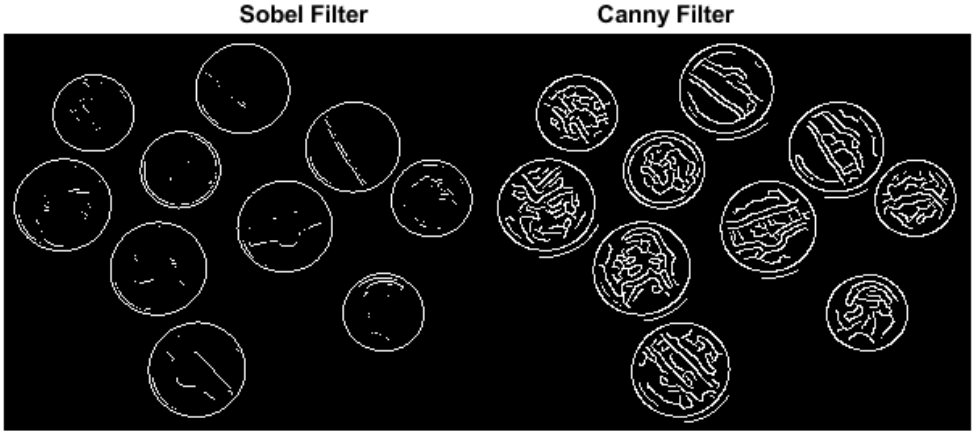
This example shows how to detect edges in an image using both the Canny edge detector and the Sobel edge detector. First read image and display it.

```
I = imread('coins.png');
imshow(I)
```

Figure 6.8: Coin Image

Apply both the Sobel and Canny edge detectors to the image and display them for comparison.

```
BW1 = edge(I,'sobel');
       BW2 = edge(I,'canny');
       figure;
       imshowpair(BW1,BW2,'montage')
title('Sobel Filter                                    Canny Filter');
```



EXERCISES:

Apply the image processing methods discuss in this lab session on an image of your choice and write down your observations.

Lab Session 7

Introduction to OpenCV based Image Processing Tools

OpenCV

OpenCV, or Open Source Computer Vision library, started out as a research project at Intel. It's currently the largest computer vision library in terms of the sheer number of functions it holds with implementations of more than 2500 algorithms. The library has interfaces for multiple languages, including Python, Java, and C++. The first OpenCV version, 1.0, was released in 2006 and the OpenCV community has grown leaps and bounds since then.

Reading, Writing and Displaying Images

Machines see and process everything using numbers, including images and text. How do you convert images to numbers? pixel values:

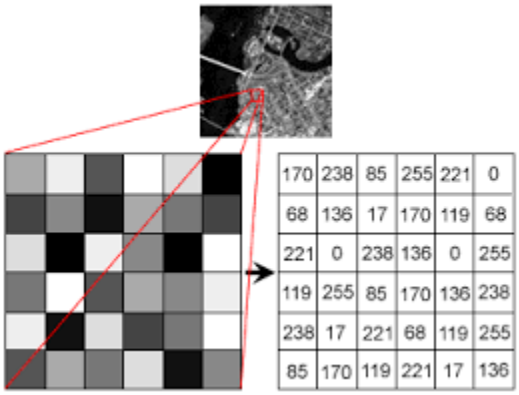


Figure 7.1: Image representation in pixel value (Grayscale Image)

Every number represents the pixel intensity at that particular location. In the above image, it has been shown that pixel values for a grayscale image are represented by one value i.e. the intensity of the black color at that location.

For color images, there are multiple values for a single pixel. These values represent the intensity of respective channels – Red, Green and Blue channels for RGB images, for instance.

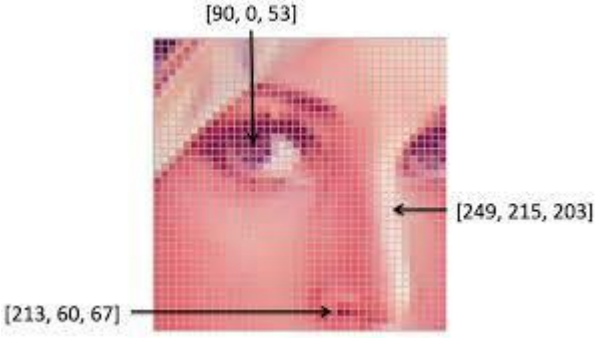


Figure 7.2: Image representation in pixel value (RGB Image)

The following code snippet help to import an image into our machine using OpenCV.

```
import the libraries
   import numpy as np
   import matplotlib.pyplot as plt
   import cv2
   %matplotlib inline
#reading the image
   image = cv2.imread('index.png')
   image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
#plotting the image
   plt.imshow(image)
#saving image
   cv2.imwrite('test_write.jpg',image)
```

```
#import the required libraries
  import numpy as np
  import matplotlib.pyplot as plt
  import cv2
  %matplotlib inline
  image = cv2.imread('index.jpg')
  #converting image to Gray scale
  gray_image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
  #plotting the grayscale image
  plt.imshow(gray_image)
  #converting image to HSV format
  hsv_image = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
  #plotting the HSV image
  plt.imshow(hsv_image)
```

Figure 7.3: RGB Filtered Image

Simple Image Thresholding

Thresholding is an image **segmentation** method. It compares pixel values with a threshold value updates it accordingly. OpenCV supports multiple variations of thresholding. A simple thresholdi function can be defined like this:

Figure 7.4: An RGB Image

Changing Color Spaces

A color space is a protocol for representing colors in a way that makes them easily reproducible. We know that grayscale images have single pixel values and color images contain 3 values for each pixel – the intensities of the Red, Green and Blue channels.

Most computer vision use cases process images in RGB format. However, applications like video compression and device independent storage – these are heavily dependent on other color spaces, like the Hue-Saturation-Value or HSV color space.

As you understand a RGB image consists of the color intensity of different color channels, i.e. the intensity and color information are mixed in RGB color space but in HSV color space the color and intensity information are separated from each other. This makes HSV color space more robust to lighting changes.

OpenCV reads a given image in the BGR format by default. So, you'll need to change the color space of your image from BGR to RGB when reading images using OpenCV. Let's see how to do that:

```
if Image(x,y) > threshold , Image(x,y) = 1
        otherswise, Image(x,y) = 0
```

Thresholding can only be applied to grayscale images.

A simple application of image thresholding could be dividing the image into it's foreground and background.

```
#importing the required libraries
import numpy as np
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
```

```
#here 0 means that the image is loaded in gray scale format
gray_image = cv2.imread('index.png',0)
ret,thresh_binary =
cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)
ret,thresh_binary_inv =
cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY_INV)
ret,thresh_trunc =
cv2.threshold(gray_image,127,255,cv2.THRESH_TRUNC)
ret,thresh_tozero =
cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO)
ret,thresh_tozero_inv =
cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO_INV)
#DISPLAYING THE DIFFERENT THRESHOLDING STYLES
names = ['Oiriginal Image','BINARY', 'THRESH_BINARY_INV',
'THRESH_ TRUNC', 'THRESH_TOZERO', 'THRESH_TOZERO_INV']
images = gray_image ,thresh_binary ,thresh_binary _inv, thresh_
trunc, thresh_tozero,thresh_tozero_inv
for i in range(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
    plt.title(names[i])
    plt.xticks([]),plt.yticks([])

plt.show()
```

Simple Image Thresholding

Thresholding is an image **segmentation** method. It compares pixel values with a threshold value and updates it accordingly. OpenCV supports multiple variations of thresholding. A simple thresholding function can be defined like this:

```
if Image(x,y) > threshold , Image(x,y) = 1
otherswise, Image(x,y) = 0
```

Thresholding can only be applied to grayscale images.

A simple application of image thresholding could be dividing the image into it's foreground and background.

```
#importing the required libraries
import numpy as np
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
#here 0 means that the image is loaded in gray scale format
gray_image = cv2.imread('index.png',0)
ret,thresh_binary =
cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)
ret,thresh_binary_inv = cv2.threshold(gray_image ,127,255, cv2.
THRESH_BINARY_INV)
ret,thresh_trunc = cv2.threshold(gray_image,127,255,cv2
```

```
.THRESH_TRUNC)
ret,thresh_tozero = cv2.threshold(gray_image,127,255,cv2
.THRESH_TOZERO)
ret,thresh_tozero_inv = cv2.threshold(gray_image,127,255,cv2
.THRESH_TOZERO_INV)
#DISPLAYING THE DIFFERENT THRESHOLDING STYLES
names = ['Oiriginal Image','BINARY','THRESH_BINARY_INV',
'THRESH_TRUNC' ,'THRESH_TOZERO','THRESH_TOZERO_INV']
images = gray_image,thresh_binary,thresh_binary_inv
,thresh_trunc ,thresh_tozero,thresh_tozero_inv
for i in range(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
    plt.title(names[i])
    plt.xticks([]),plt.yticks([])

plt.show()
```

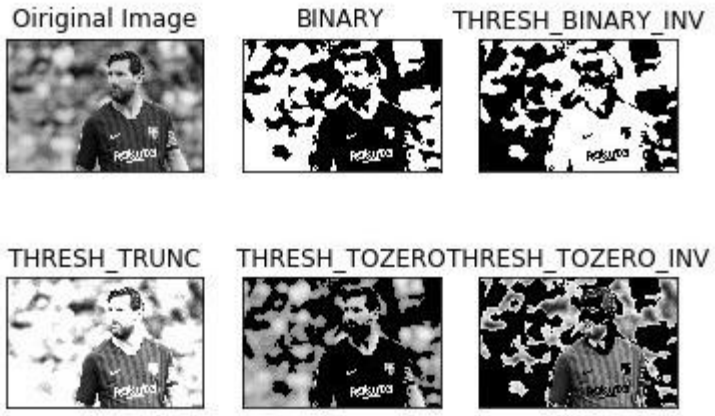


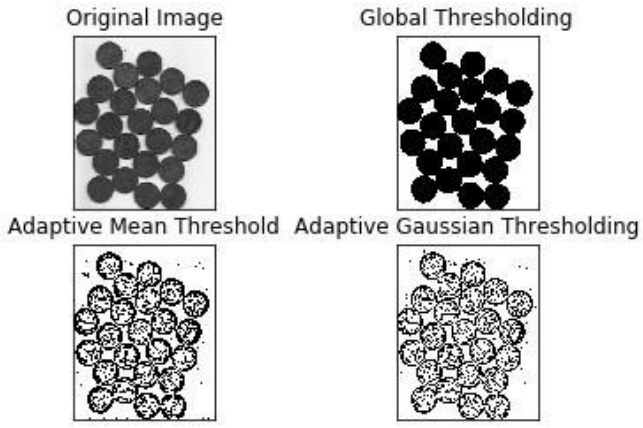
Figure 7.5: Different types of Thresholding

Adaptive Thresholding

In case of adaptive thresholding, different threshold values are used for different parts of the image. This function gives better results for images with varying lighting conditions – hence the term "adaptive".
Otsu's binarization method finds an optimal threshold value for the whole image. It works well for bimodal images (images with 2 peaks in their histogram).

```
#import the libraries
import numpy as np
import matplotlib.pyplot as plt
import cv2
%matplotlib inline
#ADAPTIVE THRESHOLDING
gray_image = cv2.imread('index.png',0)
ret,thresh_global                                                =
cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)
#here 11 is the pixel neighbourhood that is used to calculate the
```

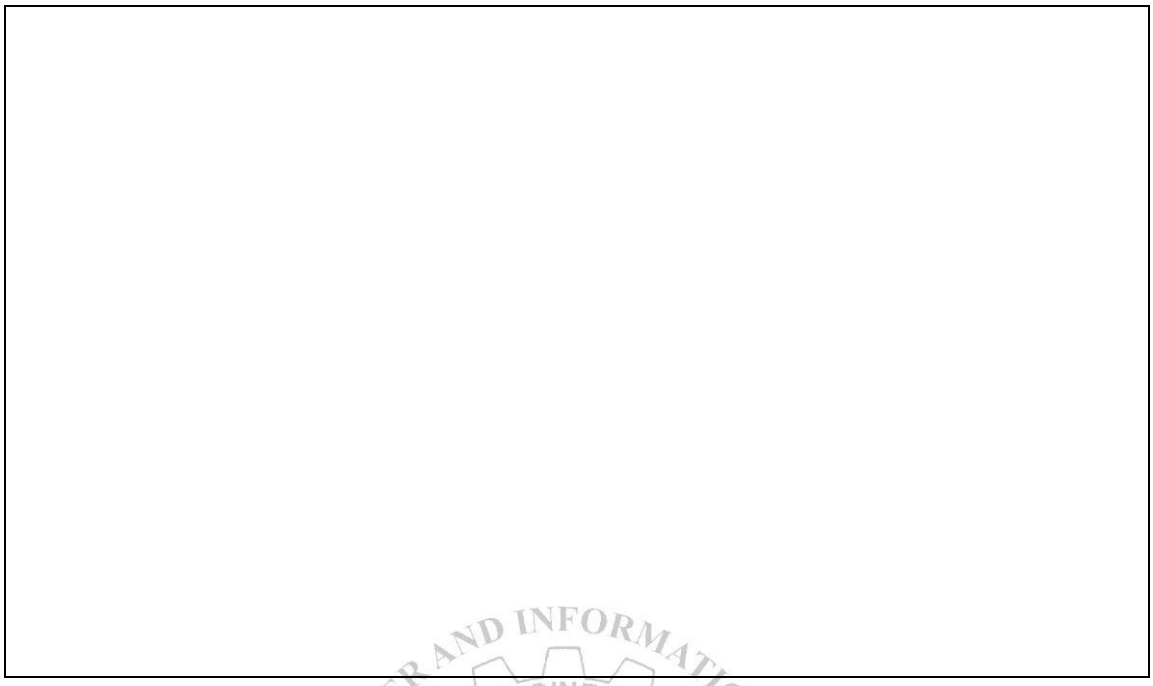
```
threshold value
thresh_mean                                                    =
cv2.adaptiveThreshold(gray_image,255,cv2.ADAPTIVE_THRESH_MEAN_C,c
v2.THRESH_BINARY,11,2)
thresh_gaussian                                                =
cv2.adaptiveThreshold(gray_image,255,cv2.ADAPTIVE_THRESH_GAUSSIAN
_C,cv2.THRESH_BINARY,11,2)
names = ['Original Image','Global Thresholding','Adaptive Mean
Threshold','Adaptive Gaussian Thresholding']
images = [gray_image,thresh_global,thresh_mean,thresh_gaussian]
for i in range(4):
    plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
    plt.title(names[i])
    plt.xticks([]),plt.yticks([])

plt.show()
```



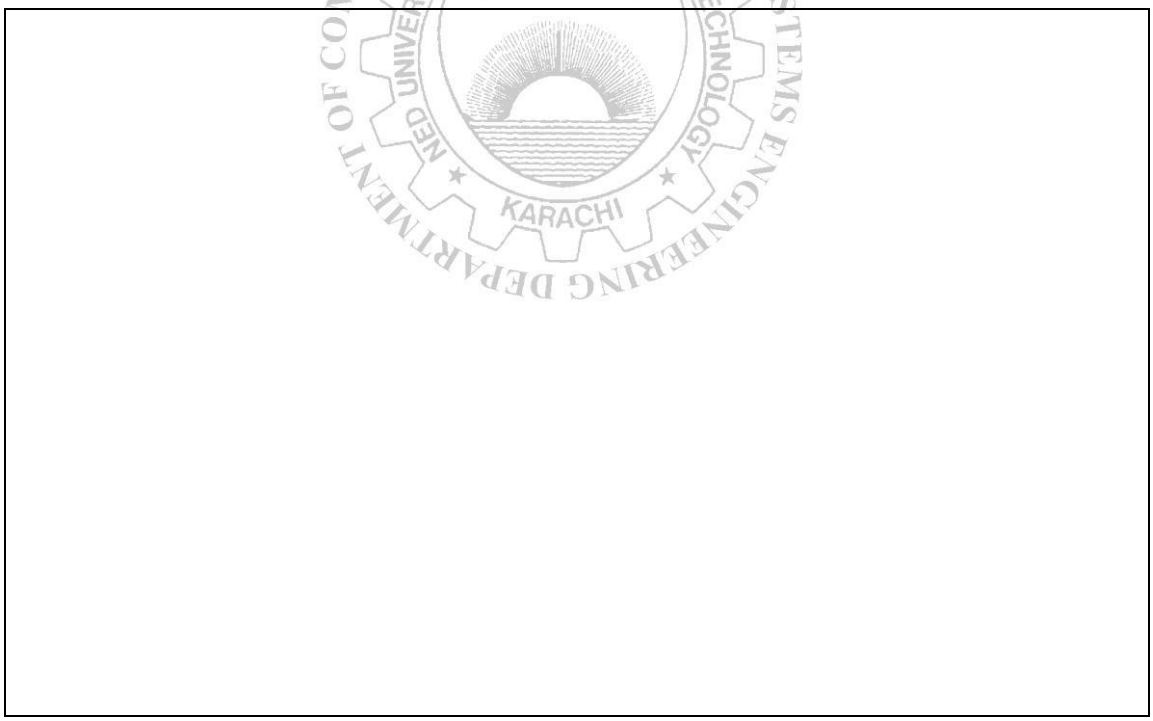
EXERCISE

Perform the below mentioned operation and attach printout for each operation output for the picture shown as under:

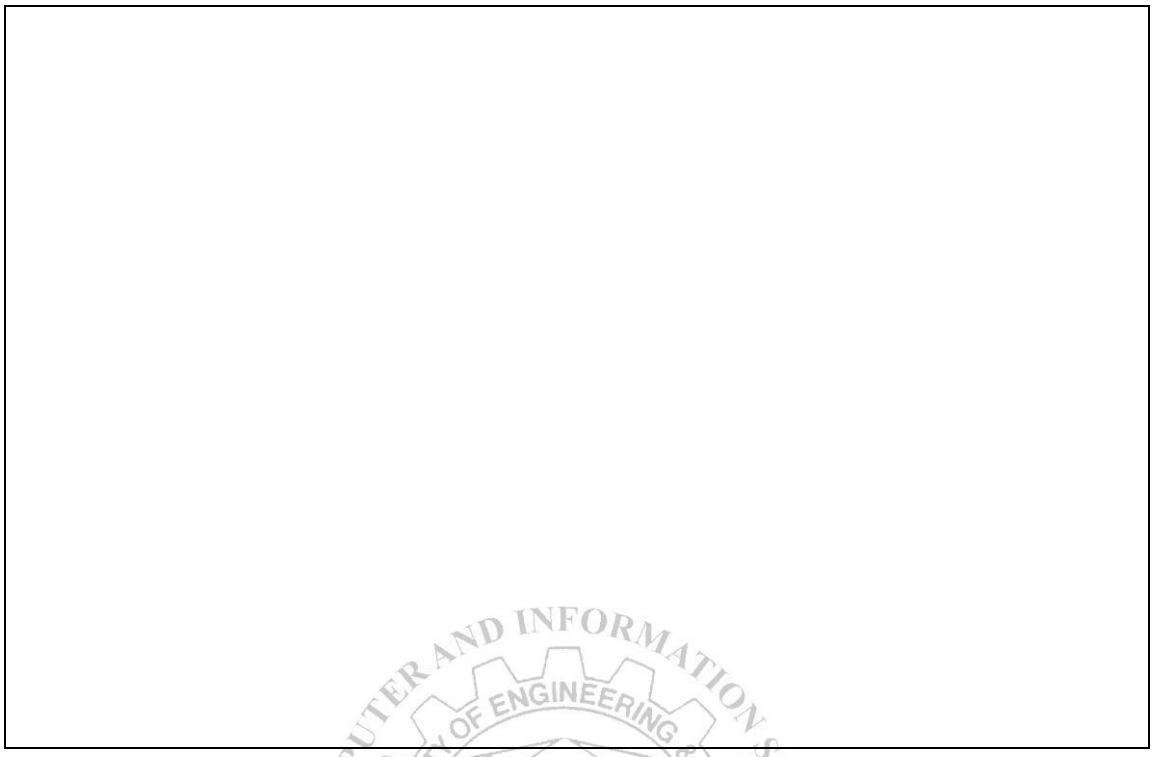
a. Plot in HSV



b. Apply static thresholding



c. Choose and apply the best result thresholding technique on the image.



Lab Session 8

Data Collection and Preprocessing for Computer Vision Applications

Basics

Data collection and preprocessing is a major step in all computer vision applications. Without proper preprocessing steps it is nearly impossible to train any computer vision algorithm properly. In this lab session, you will learn to collect data and preprocess it to properly train a computer vision algorithm.

Data Collection

For any particular application, collection of data is targeted with respect to the desired output required from the computer vision application. For example, if a particular type of car is required to be identified from an image then first large number of images of that car should be collected and stored. Let's say we need to identify SUZUKI MEHRAN car in the environment then first we need to collect ample number of images of SUZUKI MEHRAN car as shown in Fig. 8.1.

Figure 8.1: Different images of a same car (SUZUKI MEHRAN) with different orientations

For dataset, maximum number of samples ensures maximum accuracy of the computer vision applications. All these positive images should be stored in a single, unique folder which will be used for training at later stages.

Resizing Images

Machine learning models work with a fixed sized input. The same idea applies to computer vision models as well. The images we use for training our model must be of the same size.
Now this might become problematic if we are creating our own dataset by scraping images from various sources. That's where the function of resizing images comes to the fore.
Images can be easily scaled up and down using OpenCV. This operation is useful for training deep learning models when we need to convert images to the model's input shape. Different interpolation and downsampling methods are supported by OpenCV, which can be used by the following parameters:

1. **INTER_NEAREST:** Nearest neighbor interpolation
2. **INTER_LINEAR:** Bilinear interpolation
3. **INTER_AREA:** Resampling using pixel area relation
4. **INTER_CUBIC:** Bicubic interpolation over 4×4 pixel neighborhood
5. **INTER_LANCZOS4:** Lanczos interpolation over 8×8 neighborhood

OpenCV's resize function uses bilinear interpolation by default.

```
import cv2
  import numpy as np
  import matplotlib.pyplot as plt
  %matplotlib inline
  #reading the image
  image = cv2.imread('index.jpg')
  #converting image to size (100,100,3)
  smaller_image =
  cv2.resize(image,(100,100),inerpolation='linear')
  #plot the resized image
  plt.imshow(smaller_image)
```

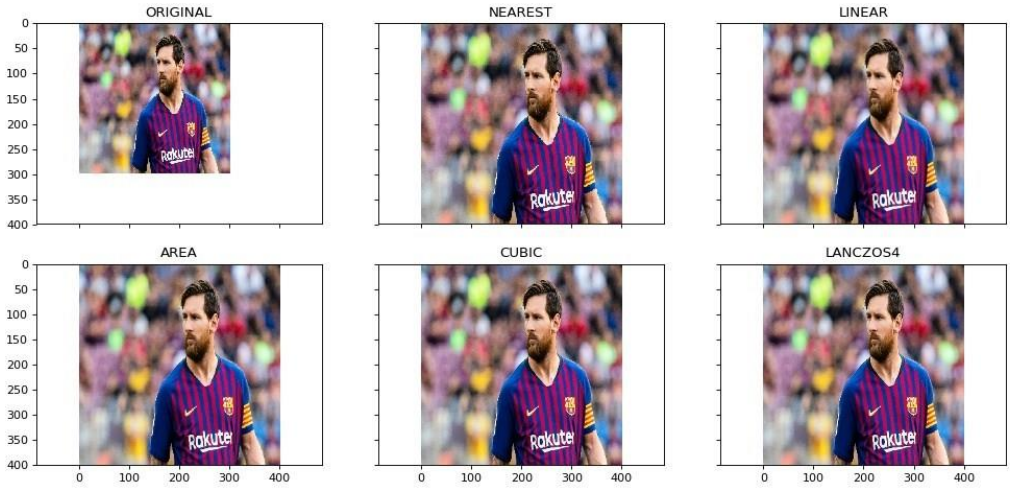


Figure 8.2: Different interpolation outputs along with the original image

Data Augmentation

Most deep learning algorithms are heavily dependent on the quality and quantity of the data. But what if you do not have a large enough dataset? Not all of us can afford to manually collect and label images.
Suppose we are building an image classification model for identifying the animal present in an image. So, both the images shown below should be classified as 'dog':

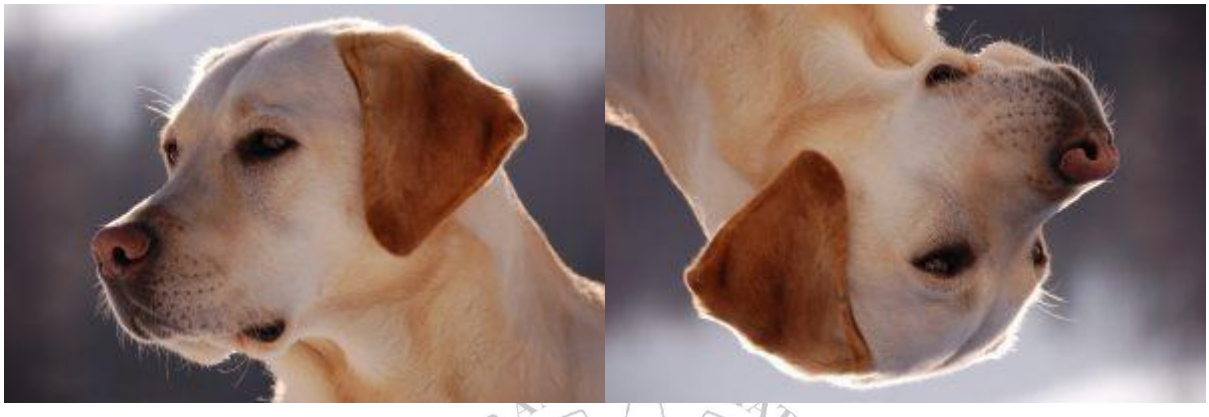


Figure 8.3: Multiple orientations of an image for data augmentation

But the model might find it difficult to classify the second image as a Dog if it was not trained on such images. So what should we do?
Let me introduce you to the technique of data augmentation. This method allows us to generate more samples for training our deep learning model. Data augmentation uses the available data samples to produce the new ones, by applying image operations like rotation, scaling, translation, etc. This makes our model robust to changes in input and leads to better generalization.

Rotation

Rotation is one of the most used and easy to implement data augmentation techniques. As the name suggests, it involves rotating the image at an arbitrary angle and providing it the same label as the original image. Think of the times you have rotated images in your phone to achieve certain angles – that's basically what this function does.

```
#importing the required libraries
  import numpy as np
  import cv2
  import matplotlib.pyplot as plt
  %matplotlib inline
  image = cv2.imread('index.png')
  rows,cols = image.shape[:2]
  #(col/2,rows/2) is the center of rotation for the
  image
  # M is the cordinates of the center
  M = cv2.getRotationMatrix2D((cols/2,rows/2),90,1)
  dst = cv2.warpAffine(image,M,(cols,rows))
  plt.imshow(dst)
```

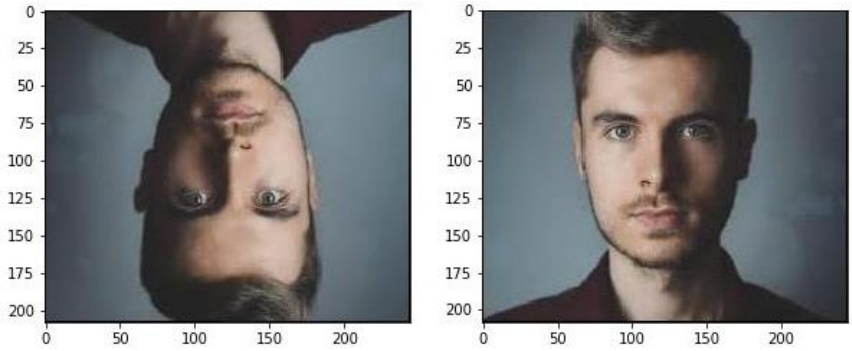


Figure 8.4: Image rotation operation

Image Translation

Image translation is a geometric transformation that maps the position of every object in the image to a new location in the final output image. After the translation operation, an object present at location (x,y) in the input image is shifted to a new position (X,Y):

```
x = x + dx
y = y + dy
```

Here, dx and dy are the respective translations along different dimensions.
Image translation can be used to add shift invariance to the model, as by tranlation we can change the position of the object in the image give more variety to the model that leads to better generalizability which works in difficult conditions i.e. when the object is not perfectly aligned to the center of the image.
This augmentation technique can also help the model correctly classify images with partially visible objects. Take the below image for example. Even when the complete shoe is not present in the image, the model should be able to classify it as a Shoe.

Figure 8.5: Image Translation as data augmentation technique

This translation function is typically used in the image pre-processing stage. Check out the below code to see how it works in a practical scenario:

```
#importing the required libraries
  import numpy as np
  import cv2
  import matplotlib.pyplot as plt
  %matplotlib inline
  #reading the image
  image = cv2.imread('index.png')
  #shifting the image 100 pixels in both dimensions
  M = np.float32([[1,0,-100],[0,1,-100]])
  dst = cv2.warpAffine(image,M,(cols,rows))
  plt.imshow(dst)
```

EXERCISES:

Develop a dataset for a particular object images as recommended by the lab instructor. Resize and Label all image properly to match the machine learning algorithm requirements. Increase the dataset by using Data augmentation techniques. The final dataset must contain 100 images, from which 60 must be retrieved from internet and 40 are generated using data augmentation techniques.

Object for Image collection: _____

Number of Images retrieved from the internet: _____

Number of Classes to be identified (Cats and Dogs are two classes): _____

Paste the printout of the data set images in a single image file here.

Lab Session 9

A Practical Computer Vision Application

In this lab session, you will learn to design first complete computer vision application from scratch. The application selected for the implementation is Face detection.

Face Detection

Face detection is a technique that identifies or locates human faces in digital images. A typical example of face detection occurs when we take photographs through our smartphones, and it instantly detects faces in the picture. Face detection is different from Face recognition. Face detection detects merely the presence of faces in an image while facial recognition involves identifying whose face it is. In this article, we shall only be dealing with the former.
Face detection is performed by using classifiers. A classifier is essentially an algorithm that decides whether a given image is positive (face) or negative (not a face). A classifier needs to be trained on thousands of images with and without faces. Fortunately, OpenCV already has two pre-trained face detection classifiers, which can readily be used in a program. The two classifiers are:

- Haar Classifier and
- Local Binary Pattern (LBP) classifier.

In this lab session, however, we will only discuss the Haar Classifier.

Haar feature-based cascade classifiers

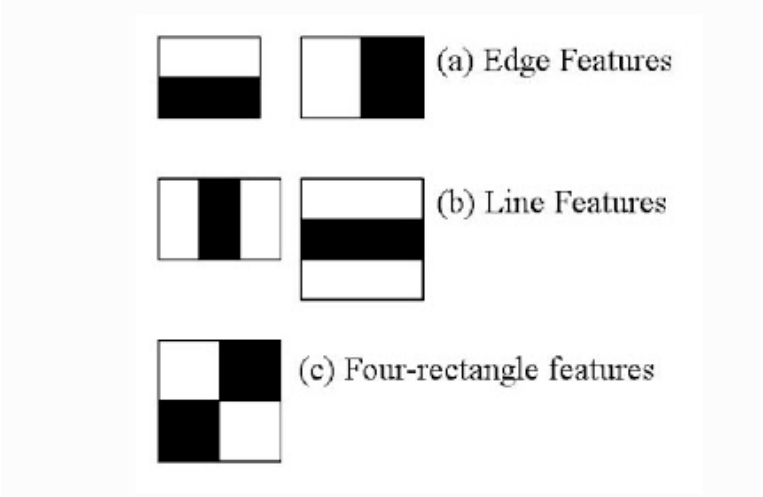
Haar-like features are digital image features used in object recognition. They owe their name to their intuitive similarity with Haar wavelets and were used in the first real-time face detector. **Paul Viola** and **Michael Jones** in their paper titled "Rapid Object Detection using a Boosted Cascade of Simple Features" used the idea of Haar-feature classifier based on the Haar wavelets. This classifier is widely used for tasks like face detection in computer vision industry.
Haar cascade classifier employs a machine learning approach for visual object detection which is capable of processing images extremely rapidly and achieving high detection rates. This can be attributed to three main reasons:

- Haar classifier employs ***'Integral Image'*** concept which allows the features used by the detector to be computed very quickly.
- The learning algorithm is based on ***AdaBoost***. It selects a small number of important features from a large set and gives highly efficient classifiers.
- More complex classifiers are combined to form a '***cascade***' which discard any non-face regions in an image, thereby spending more computation on promising object-like regions.

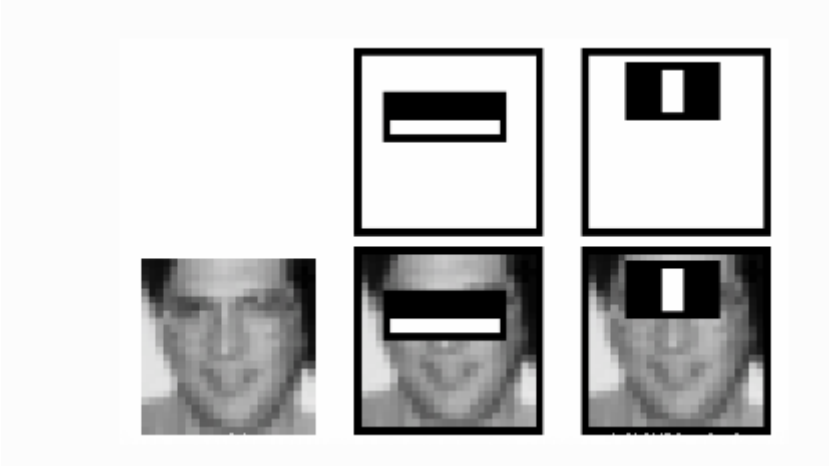
Let us now try and understand how the algorithm works on images in steps:

1. 'Haar features' extraction

After the tremendous amount of training data (in the form of images) is fed into the system, the classifier begins by extracting Haar features from each image. Haar Features are kind of convolution kernels which primarily detect whether a suitable feature is present on an image or not. Some examples of Haar features are mentioned below:



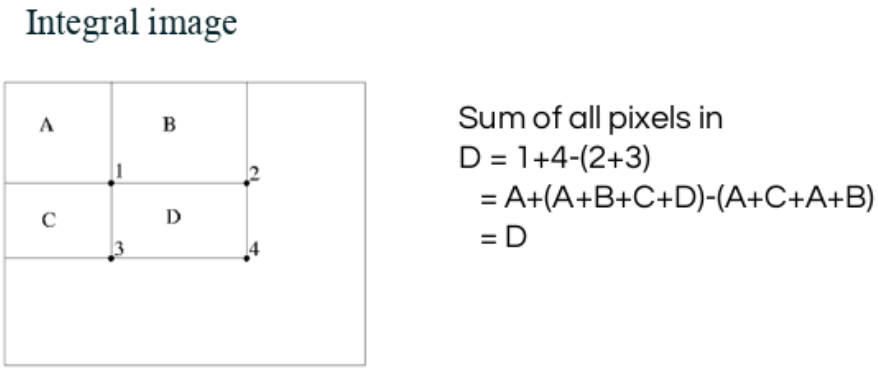
These Haar Features are like windows and are placed upon images to compute a single feature. The feature is essentially a single value obtained by subtracting the sum of the pixels under the white region and that under the black. The process can be easily visualized in the example below.



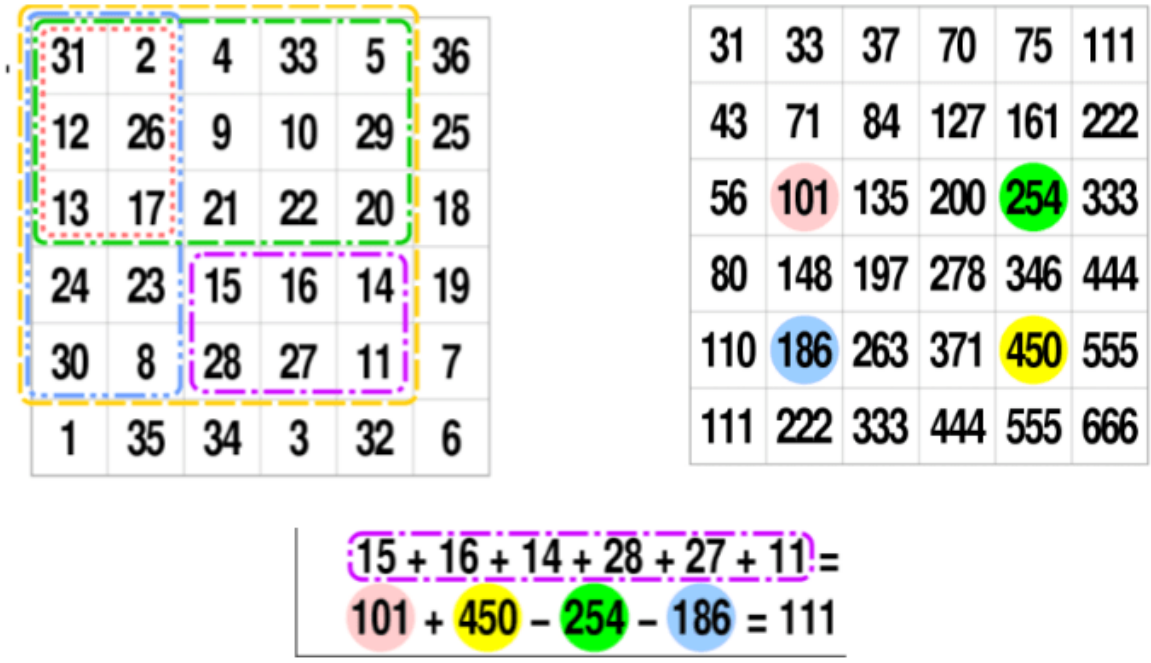
For demonstration purpose, let's say we are only extracting two features, hence we have only two windows here. The first feature relies on the point that the eye region is darker than the adjacent cheeks and nose region. The second feature focuses on the fact that eyes are kind of darker as compared to the bridge of the nose. Thus, when the feature window moves over the eyes, it will calculate a single value. This value will then be compared to some threshold and if it passes that it will conclude that there is an edge here or some positive feature.

2. 'Integral Images' concept

The algorithm proposed by Viola Jones uses a 24X24 base window size, and that would result in more than 180,000 features being calculated in this window. Imagine calculating the pixel difference for all the features? The solution devised for this computationally intensive process is to go for the **Integral Image** concept. The integral image means that to find the sum of all pixels under any rectangle, we simply need the four corner values.



This means, to calculate the sum of pixels in any feature window, we do not need to sum them up individually. All we need is to calculate the integral image using the 4 corner values. The example below will make the process transparent.



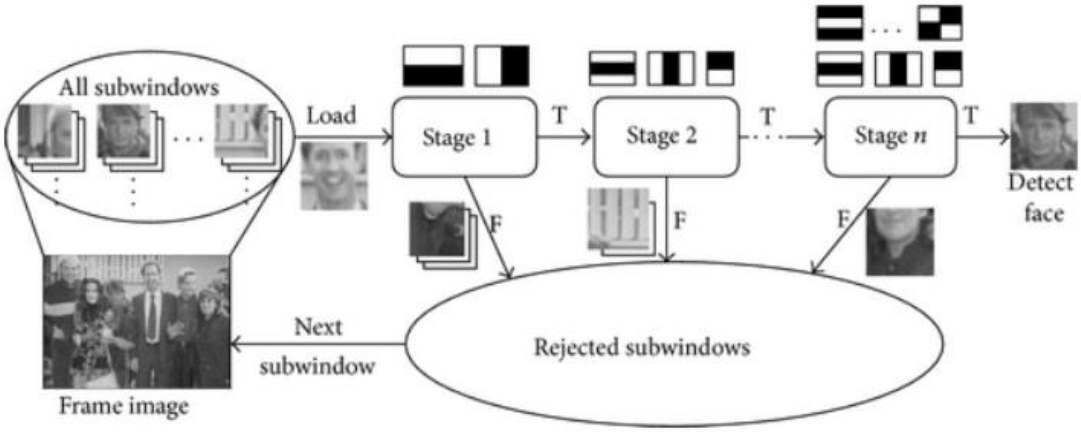
3. 'Adaboost': to improve classifier accuracy

As pointed out above, more than 180,000 features values result within a 24X24 window. However, not all features are useful for identifying a face. To only select the best feature out of the entire chunk, a machine learning algorithm called **Adaboost** is used. What it essentially does is that it selects only those features that help to improve the classifier accuracy. It does so by constructing a strong classifier which is a linear combination of a number of weak classifiers. This reduces the amount of features drastically to around 6000 from around 180,000.

4. Using 'Cascade of Classifiers'

Another way by which Viola Jones ensured that the algorithm performs fast is by employing a **cascade of classifiers**. The cascade classifier essentially consists of stages where each stage consists of a strong classifier. This is beneficial since it eliminates the need to apply all features at once on a window. Rather, it groups the features into separate sub-windows and the classifier at each stage determines whether or not the sub-window is a face. In case it is not, the sub-window is discarded along with the

features in that window. If the sub-window moves past the classifier, it continues to the next stage where the second stage of features is applied. The process can be understood with the help of the diagram below.



Face Detection with OpenCV-Python

Now we have a fair idea about the intuition and the process behind Face recognition. Let us now use OpenCV library to detect faces in an image.

Load the necessary Libraries

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading the image to be tested in grayscale

We shall be using the image below:

```
#Loading the image to be tested
test_image = cv2.imread('data/baby1.jpg')

#Converting to grayscale
test_image_gray = cv2.cvtColor(test_image, cv2.COLOR_BGR2GRAY)

# Displaying the grayscale image
plt.imshow(test_image_gray, cmap='gray')
```

Since we know that OpenCV loads an image in BGR format, so we need to convert it into RBG format to be able to display its true colors. Let us write a small function for that.

```
def convertToRGB(image):

    return cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

Haar cascade files

OpenCV comes with a lot of pre-trained classifiers. For instance, there are classifiers for smile, eyes, face, etc. These come in the form of xml files and are located in the opencv/data/haarcascades/ folder. Download the xml files and place them in the data folder in the same working directory as the jupyter notebook.

Loading the classifier for frontal face

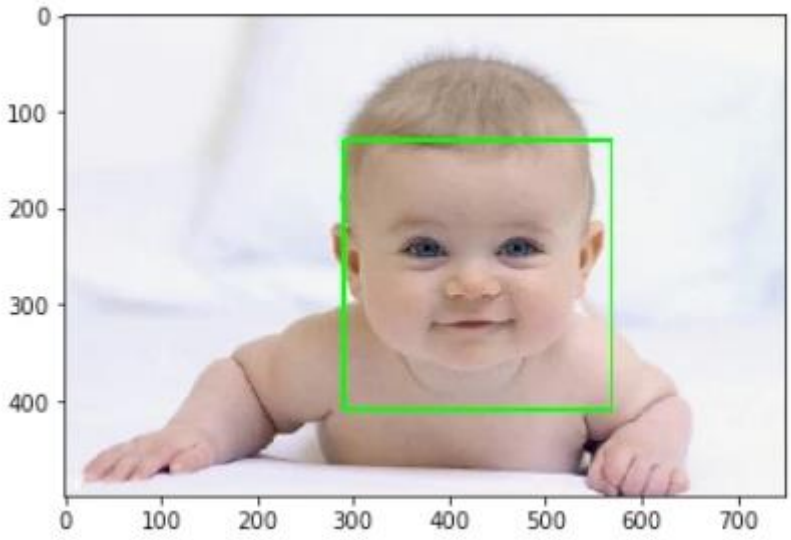
haar_cascade_face= cv2.CascadeClassifier ('data / haarcascade / haarcascade _ frontalface _ default.xml')

Face detection

We shall be using the **detectMultiscale** module of the classifier. This function will return a rectangle with coordinates(x,y,w,h) around the detected face. This function has two important parameters which have to be tuned according to the data.

- **scalefactor** In a group photo, there may be some faces which are near the camera than others. Naturally, such faces would appear more prominent than the ones behind. This factor compensates for that.
- **minNeighbors** This parameter specifies the number of neighbors a rectangle should have to be called a face.

```
#convert image to RGB and show image

plt.imshow(convertToRGB(test_image))
```



Here, it is. We have successfully detected the face of the baby in the picture. Let us now create a generalized function for the entire face detection process.

Face Detection with generalized function

```
def detect_faces(cascade, test_image, scaleFactor = 1.1):
    # create a copy of the image to prevent any changes to the
original one.
    image_copy = test_image.copy()

    #convert the test image to gray scale as opencv face detector
expects gray images
    gray_image = cv2.cvtColor(image_copy, cv2.COLOR_BGR2GRAY)

    # Applying the haar classifier to detect faces
    faces_rect = cascade.detectMultiScale(gray_image,
scaleFactor=scaleFactor, minNeighbors=5)

    for (x, y, w, h) in faces_rect:
        cv2.rectangle(image_copy, (x, y), (x+w, y+h), (0, 255, 0),
15)

    return image_copy
```

Testing the function on new image

Test computer vision program mentioned above on the following images:

(Hint: Search baby images from google and download the ones shown here.)


EXERCISE

Implement the above mentioned Haar-cascade algorithm on another dataset recommended by lab instructor.

Lab Session 10

Learning basic concepts of frames and inheritance for Expert System Programming

flex

flex is an expressive and powerful expert system toolkit which supports frame-based reasoning with inheritance, rule-based programming and data-driven procedures fully integrated within a logic programming environment, and contains its own English-like Knowledge Specification Language (KSL).

flex goes beyond most expert system shells in that it employs an open architecture and allows you to access, augment and modify its behavior through a layer of access functions. Because of this, *flex* is often referred to as an AI toolkit. The combination of *flex* and Prolog i.e. a hybrid expert system toolkit with a powerful general-purpose AI programming language, results in a functionally rich and versatile expert system development environment where the developers can fine tune and enhance the built-in behavior mechanisms that suits their own specific requirements.

flex appeals to various groups of developers; expert systems developers who want to deliver readable and maintainable knowledge-bases, advanced expert system builders who want to incorporate their own controls, AI programmers who want access to a high-level language-based product and Prolog programmers who require extra functionality and structures.

Expert Systems

Expert systems (or knowledge-based systems) allow the scarce and expensive knowledge of experts to be *explicitly* stored into computer programs and made available to others who may be less experienced. They range in scale from simple rule-based systems with flat data to very large scale, integrated developments taking many person-years to develop. They typically have a set of **`if-then`** rules which forms the *knowledge base*, and a dedicated *inference engine*, which provides the execution mechanism. This contrasts with conventional programs where domain knowledge and execution control are closely intertwined such that the knowledge is *implicitly* stored in the program. This explicit separation of the knowledge from the control mechanism makes it easier to examine knowledge, incorporate new knowledge and modify existing knowledge.

Forward Chaining

Forward chaining *production rules* in *flex* follow the classical **`if-then`** rule format. Forward chaining is data-driven and is very suitable for problems which involve too many possible outcomes to check by backward chaining, or where the final outcome is not known. The forward chaining inference engine cycles through the current rule agenda looking for rules whose **`if`** conditions can be satisfied, and selects a rule to use or *fire* by executing its **`then`** part. This typically side effects data values, which means that a different set of rules now have their conditions satisfiable.

Flex extends the classical production rule with an optional explanation facility and dynamic scoring mechanism for resolving conflicts during rule selection. Rules can have multiple conclusions or actions (either positive or negative) in their **`then`** part.

The rule selection and agenda update algorithms of the forward chaining engine are flexible, with many built-in algorithms and the option of applying user-defined algorithms.

Backward Chaining

Backward chaining rules, which correspond closely to Prolog predicates, are called *relations* in *flex*.

They have a single conclusion that is true, if all the conditions can be proven. Backward chaining is often referred to as goal-driven, and is closely linked to the notion of provability.

Search

Search is one of the key characteristics of expert systems. There are normally many ways of combining or chaining rules together with data to infer new conclusions. How to examine only the relevant part of this *search space* is a serious consideration with regard to efficiency. The ordering of rules, the provision of meta-rules (rules about which rules to use) and conflict-resolution schemes are all ways of helping us produce a sensible *search tree* which we can investigate. Prolog-based systems tend to use a depth-first strategy, whereby a certain path is fully explored by checking related paths, combined with *backtracking* to go back and explore other possibilities when a dead-end is reached.

Frames and Inheritance

Frame hierarchies are similar to object-oriented hierarchies. They allow data to be stored in an abstract manner within a nested hierarchy with common properties automatically inherited through the hierarchy. This avoids the unnecessary duplication of information, simplifies code and provides a more readable and maintainable system.

Each **`frame`** or **`instance`** has a set of slots that contain attributes describing the frame's characteristics. These slots are analogous to fields within records (using database terminology) except that their expressive power is greatly extended.

Frames **`inherit`** attribute-values from other frames according to their position in the frame hierarchy. This inheritance of characteristics is automatic, but can be controlled using different built-in algorithms.

Questions and Answers

flex has a built-in **`question`** and **`answer`** sub-system that allows final applications to query the user for additional input via interactive dialogs. These screens can be simple pre-defined ones, or complex, sophisticated screens constructed using Prolog's own screen handling facilities and then attached to the question and answer sub-system.

Explanations

flex has a built-in explanation system which supports both *how* and *why* explanations. Explanations can be attached to both rules and questions using simple **`because`** clauses.

Data-driven Programming

flex offers special procedures which can be attached to collections of frames, individual frames or slots within frames. These procedures remain dormant until activated by the accessing or updating of the particular structure to which they have been attached. There are four different types of data-driven procedures available within *flex*: **`launches`**, **`demons`**, **`watchdogs`**, and **`constraints`**.

Knowledge Specification Language

flex has its own expressive English-like Knowledge Specification Language (KSL) for defining rules, frames and procedures. The KSL enables developers to write simple and concise statements about the expert's world and produce virtually self-documenting knowledge-bases which can be understood and maintained by non-programmers. The KSL supports mathematical, Boolean and conditional expressions and functions along with set abstractions; furthermore, the KSL is extendable through synonyms and templates. By supporting both logical and global *variables* in rules, *flex* avoids

unnecessary rule duplication and requires fewer rules than most other expert systems.

How to Start

To start working in flex, you have to type the following in the console window.

```
ensure_loaded(system(flexenv)).
```

Frames and Inheritance

In this chapter we describe the frame sub-system of *flex*. This includes the form and content of individual frames, how frames are linked together to form a frame hierarchy, and how values are inherited through that hierarchy.

What is a Frame?

A frame is similar to an object and is a complex data structure which provides a useful way of modeling real-world data objects.

Frames are analogous to records within a database but are far more powerful and expressive. Each individual frame has a name by which it is referred, details of its parent(s) frame, and a collection of slots or attributes (similar to fields within records) which will contain values or pointers to values. Slot values can be explicitly defined locally, or implicitly *inherited* from an ancestor frame further up the hierarchy.

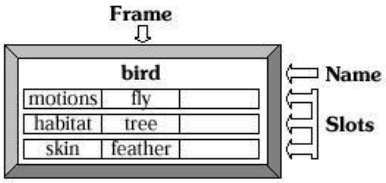


Figure 10.1

flex has its own language (for representing frames and other constructs) called the Knowledge Specification Language. For example, the KSL code for the above frame could be:

```
frame bird
   default skin is feather and
   default habitat is a tree and
   default motions are { fly } .
```

Each slot has three principal components:

- attribute name - such as habitat, describing the concept
- default value - the default value, to be used when there is no current value
- current value - the current value for the attribute

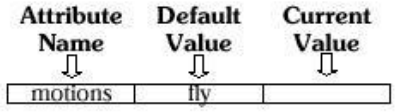


Figure 6.2

A frame can be viewed as a dynamic array having three columns *(Attribute Nam*e, *Default Value* and *Current Valu*e) and an arbitrary number of rows, one for each slot.

It is important to note the difference between *default* and *current* values, since some *flex* operations work on only on *current* values.

The following example is an illustration of a frame representing the concept of a jug.

Figure 6.3

Slots may or may not have values. For example, there is a current but not a default value for the `position` of the `jug`, a default but not a current value for the `capacity` of the `jug`, and both a default and a current value for the contents of the jug. The default value for a slot is used *only* in the absence of a current value for that slot.

When a frame is declared in the KSL, the initial default values of its attributes may be declared, as in the above example of the frame `bird`.

However, additional slots may be added dynamically simply by referring to them and giving them a value. For example, the above `jug` frame may be declared in KSL as:

```
frame jug
   default capacity is 15 and
   default contents is 0.
```

Its position slot may then be created and its `contents` updated as follows, using a KSL action (described later).

```
action jug_update ;
   do the contents of the jug becomes 7.5 and
   the position of the jug becomes upright.
```

There are no restrictions on what terms can be used as the default or current values of slots. They can be any valid Prolog term. They can be calculations (or *access functions*), which are performed whenever you need the slotvalue.

```
frame box
   default width is 10 and
   default depth is 5 and
   default volume is its width times its depth .
```

Linking Frames

Frames can be linked to each other, it enables a *frame hierarchy* to be established.

Let us consider a small section of the animal kingdom as below:

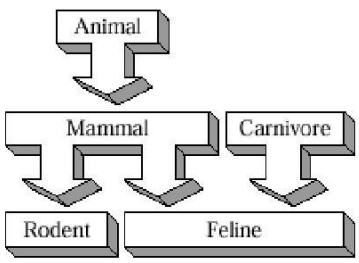


Figure 6.4

An arrow pointing from one frame to another indicates a parent-frame to child-frame link in the hierarchy.

The KSL frame declarations for the above diagram are as follows:

```
frame animal .

frame carnivore .

frame mammal is an animal
   default blood is warm and
   default habitat is land .

frame rodent is a kind of mammal
   default habitat is sewer .

frame feline is a mammal, carnivore .
```

Creating an Instance of a Frame

So far we have discussed the use of frames to represent general (static) objects such as mammals, felines and rodents. However, frames may also represent specific (dynamic) instances of objects such as `Sylvester` (a well-known cat) or `Sammy` (my cat).

In formal terms there is very little to distinguish a *frame* representing a class of objects from an *instance* representing a specific instance of the frame. Instances appear as leaf nodes in the frame hierarchy and can have only one single parent-frame. In addition, instances may only contain *current* values in their slots; they may not have default values declared.

Example:

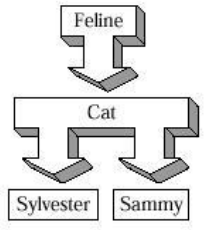


Figure 6.5

The instances are represented by a box without a shadow.
The KSL representation of the above is as follows.

```
frame feline is a mammal, carnivore
   default legs are 4 .

frame cat is a feline
   default habitat is house and
   default meal is kit_e_kat .

instance sylvester is a kind of cat .

instance sammy is an instance of cat .
```

Here, by default, both `sylvester` and sammy will live in a `house`, eat `kit_e_kat` and have 4 legs.

Overriding Inheritance

In our examples so far, a child-frame will automatically inherit from its parent-frames. We may wish, however, for a particular attribute to be inherited from a frame outside the hierarchy, or from a particular frame within the hierarchy, or not inherited at all.

Specialized Inheritance

In *flex* a special inheritance link may be defined that allows a specific attribute to be inherited from a specific frame.
For Example;

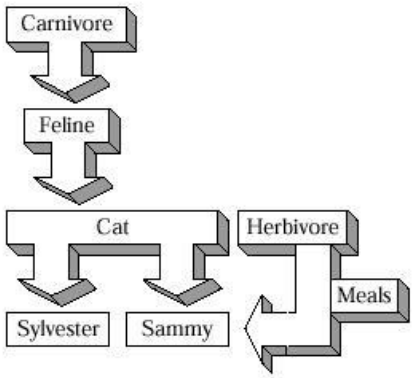


Figure 6.6

The corresponding KSL code would be:

```
instance sammy is an instance of cat ;
   inherit meal from herbivore .
```

Note that `herbivore` is not a parent of `sammy`: it only contributes the `meal` attribute.

Negative Inheritance

In *flex* the inheritance of a particular attribute for a particular frame may be suppressed.

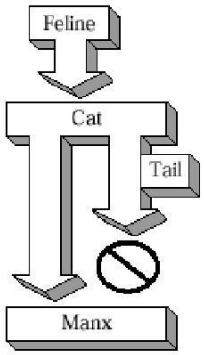


Figure 6.7

The KSL code for this is as follows:

```
frame cat
   default tail is furry .

frame manx is a cat
```

```
   do not inherit tail .
```

Attribute Chaining

Sometimes it may be convenient for an attribute to have its own *set* of values, and in this case slots may contain pointers to other frames rather than simple values.

```
frame address
   default city is 'London' .

frame employee
   default residence is an address .
```

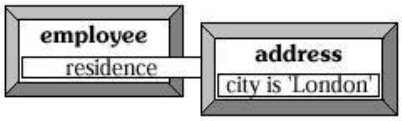


Figure 6.8

In this example, the value attached to the `residence` attribute of the `employee` frame is a pointer to another frame, namely `address`.

If we want to know the city of residence of an employee, we can refer to this in three different ways:

```
X is the residence of employee
and Y is the city of X
or

Y is the city of the residence of employee or,
```

using the operator `s as shorthand

```
Y is employee`s residence`s city
```

all of which make `London` the value of the variable Y.

For example, if we create a new employee instance called `phil`, then it will be assumed that `phil` lives in `London`.

```
instance phil is an employee .
```

If, however, Phil does not live in London, but in Glasgow, then we can reflect this with the following directive.

```
do the city of residence of phil becomes 'Glasgow'
```

This has actually set up the following structure.

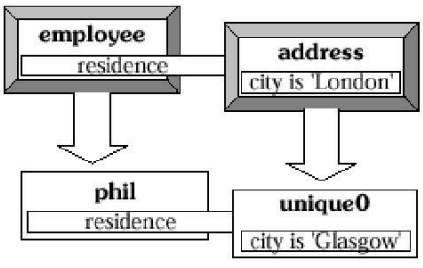


Figure 6.9

Global Variables

One special use of frames is to store global variables. These are defined as attributes of a special `frame` called `global`.

```
frame global
   default current_interest_rate is 10.3 .
```

This creates a global variable called `current_interest_rate` which may then be referred to by any KSL statement. The values of global variables may be updated at run-time. Global variables are also used to store the response to a *flex* `question` – see the chapter on Questions.

Frame Relationships

In its default setting, the only relationship between frames is the AKO (a-kind- of) hierarchy which defines how values are to be inherited. In general, though, it would be of great benefit to be able to define other relationships between frames, such as all tigers can hunt humans.

```
frame tiger .
frame human .
relation can_hunt( tiger, human ) .
```

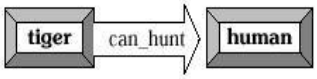


Figure 6.10

In its present form, the extension of the `can_hunt/2` relation contains only a single tuple, namely the pair <`tiger,human`>. If we were to pose the Prolog query …

```
?- prove( can_hunt( X, Y ) ) .
```

there would be a single solution which binds the identifier `tiger` to the variable X, and binds the identifier `human` to the variable Y. (`prove/1` is a built-in *flex* predicate, described later.) Now consider two particular instances of tiger and human.

```
instance shere_khan is a tiger .
instance mowgli is a human .
```

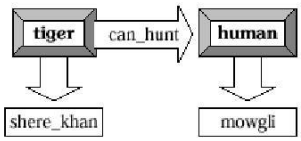


Figure 6.11

The answers to our query above will remain the same, namely only a single solution. This is because the underlying logic only allows unification between objects which have the same name (i.e. pattern-matching). The query

```
?- prove( can_hunt( shere_khan, mowgli ) ) .
```

would fail because `shere_khan` does not match `tiger`, and furthermore because `mowgli` does not match `human`. The *flex* system allows the underlying logic to be changed from one involving unification to one involving inheritance. That is, although …

`tiger` *does not match* `shere_khan`

`human` *does not match* `mowgli`

with an inheritance logic we can show that …

`tiger`	*is an ancestor of*	`shere_khan`	*in the frame hierarchy*
`human`	*is an* ancestor *of*	`mowgli`	*in the frame hierarchy*

and as such we can conclude that the `tiger shere_khan` can hunt the `human mowgli`.
The underlying logic can be changed by issuing the Prolog command …

```
?- new_logic( inherit ) .
```

It should be noted at this point that there is a general overhead involved in changing from a unification-based logic to an inheritance-based logic. That is, every procedure invocation will involve data lookup rather than direct pattern-matching. It should also be noted that the inheritance-based logic can only be used for *checking* given values, and not for generating instances of relationships. For example, it can check that `shere_khan` can hunt `mowgli`, but will not be able to generate it.

The Anatomy of a *flex* Program

This lab session describes the basic composition of a *flex* program using the Knowledge Specification Language (KSL).

A *flex* program comprises a series of sentences written in the KSL (Knowledge Specification Language). Each sentence starts with a KSL keyword and ends with a full stop. These sentences are compiled into Prolog clauses by the *flex* compiler.

A KSL sentence begins with one of the following keywords:

> **Action, constraint, data, demon, do, frame, function, group, instance, launch, question, relation, rule, ruleset, synonym, template, watchdog**

Note that a KSL program does not have to use forward chaining, and may consist entirely of **relation**s and **action**s, which are the equivalent of backward chaining Prolog programs. For example, the following Prolog program.

```
sibling( X, Y ):-
parent( Z, X ),
parent( Z, Y ).
```

may be written using the KSL as follows.

```
relation sibling( X, Y ) if
parent( Z, X )
and parent( Z, Y ).
```

Either of these may be called from the Prolog command line, e.g.

```
?- sibling( harriet, S ) .
```

However, to use the *flex* forward chaining engine a minimal *flex* program contains at least one of frame, rule, ruleset, action.

The **frames** describe the data structures. The frames have *slots* (sometimes called *attributes*), which are like fields in a conventional record structure. The rules defined in a *flex* program manipulate the

data contained in these frames.

A *flex* **rule** consists of a set of conditions and some actions to be performed if the conditions are satisfied. A rule is said to *fire* if its conditions are satisfied. A *flex* **ruleset** declares the names of the rules to be used for the current problem.

The basic mechanism of the *flex* forward chaining engine is to go through the current set of rules, testing the conditions, until a rule is found whose conditions are satisfied, and its actions are then performed. This cycle repeats until no more rules can be fired, i.e. there are no rules whose conditions can be satisfied. It is possible to specify other termination criteria, and to specify exactly which rules should be considered, in what order, and how they should be reordered for each cycle of the forward chaining engine.

The *flex* forward chaining engine may be started by defining an **action**, which is similar to a Prolog program and may be run as a Prolog query.

A Simple *flex* Program

We may write a very simple *flex* program to sell cinema seats to viewers. This will demonstrate the essential components of a forward chaining *flex* program, and give a flavour of the KSL. Note that because this is KSL code it should be compiled in a file or window with the extension `.KSL`.

The frames

First we define the data structures, which mean declaring a `frame` for a cinema and a `frame` for a cinema-goer. Each will have a single slot, or attribute, which contains the number of seats for a cinema and the number of tickets required by a viewer.

```
frame cinema
default seats is 500.
frame viewer
default tickets_required is 3.
```

The rules

We will define one rule, which describes how the cinema seats will be allocated to the viewer. We will call this rule `allocate_tickets`. The condition under which the rule will be fired is simply that the viewer requires some tickets!

```
rule allocate_tickets
if the tickets_required of viewer is greater than 0 then
the seats of the cinema becomes
the seats of the cinema
minus the tickets_required of viewer and the
tickets_required of viewer becomes 0.
```

The ruleset

Next, a `ruleset` must be defined to say what rules are to be considered. We will call our ruleset `seating`; in this case we only have one rule, called `allocate_tickets`.

```
ruleset seating
contains allocate_tickets.
```

The action

Finally, to set this going we *invoke* these rules, defining an **action** to do so.

```
action go ;
do invoke ruleset seating.
```

Starting The Forward Chaining Engine

We now have a complete *flex* program. To start the *flex* forward chaining engine, we simply run the Prolog query `go` (type `go` at the Prolog command line).

```
?- go.
```

This should succeed, with the given rule firing once only (because after that the viewer's tickets required will be zero).

Displaying Results

Unfortunately, at the moment we have no way of knowing if it ran correctly, i.e. if 3 seats were subtracted from the cinema's total seats. We will add another `action` to simply write out some relevant values, called `write_values`.

Here we will use the KSL operator‘s instead of **of** for accessing the slots of frames.

```
action write_values ;
write( 'Cinema seats: ' ) and
write( cinema`s seats ) and tab(
2 ) and
write( 'Viewer tickets required: ' ) and
write( viewer`s tickets_required ) and nl.
```

Note the use of the built-in Prolog predicates `write/1`, `tab/1` and `nl/0` to write text, spaces and a new line. Any Prolog predicate may be called in this way from *flex*; its arguments will be dereferenced by the *flex* interpreter before executing (so that, for example, the term `viewer`s tickets_required` will be dereferenced to the current value of the slot `tickets_required` of the frame `viewer`).

By writing out the values before and after running *flex*, we may see that the operation has been done correctly.

```
?- restart, write_values, go, write_values.
Cinema seats: 500 Viewer tickets required: 3
Cinema seats: 497 Viewer tickets required: 0
```

Note that `restart/0` is a built-in *flex* predicate which resets slot values back to their original values.

Extending the Program

In reality there would be more than one cinema and more than one viewer, (so we would probably uses `instances`), and the forward chaining engine would continue until there were no more `viewers` with `tickets_required` values greater than 0. There would also need to be a check that the number of tickets required was less than the number of seats in the cinema.

```
rule allocate_tickets
if the viewer`s tickets_required is greater than 0 and
the cinema`s seats is greater than or equal to the
viewer`s tickets_required
then the seats of the cinema becomes the
```

```
seats of the cinema
minus the tickets_required of viewer and the
tickets_required of viewer becomes 0 .
```

A second rule could be added to inform the viewer that no seats were available.

```
rule refuse_tickets
if the viewer`s tickets_required is greater than 0 and
the cinema`s seats is less than
the viewer`s tickets_required
then write( 'Sorry - no seats left' ) and nl and the
tickets_required of viewer becomes 0 .
```

It would also be better if there was some user interaction so that at runtime we could ask how many tickets the viewer wanted - for this we could use the *flex* **question** construct.

flex and Prolog

flex is built on top of Prolog and all the functionality of Prolog is also available to the *flex* programmer. Any Prolog predicate (either built-in or user-defined) may be called from within *flex*, and its arguments will be dereferenced before being called. Conversely, any `action` or `relation` defined in the KSL may be called from Prolog as if it were a Prolog program. Applications may therefore be easily written as a mix of Prolog and *flex*. However, we also provide full access to the *flex* system from Prolog alone: you do not have to use the KSL at all. You may write *flex* programs entirely in Prolog, using the *flex* predicates listed in the *Flex Predicates* chapter of this manual. For each KSL sentence there is an equivalent set of Prolog predicates. Alternatively you may write part of your code using the Prolog *flex* predicates provided, and write part of your code using the KSL. Prolog and KSL code may be freely intermixed.

Note that you will normally have to compile KSL and Prolog code separately; KSL code should be stored in a file or window with the extension `.KSL`.

Components of the KSL

The KSL contains *terms*, *formulae*, and *sentences*.

- The *terms* describe the objects in the world being defined.
- The *formulae* are used to describe the relationships between different objects of the KSL.
- The *sentences* of the KSL are valid statements which relate the formulae and terms.

A KSL program comprises a series of sentences.

EXERCISES

1. Write the code for defining a frame named, student, with different attributes like name, father's name, AI-marks & roll no. Assign your particulars as default values.

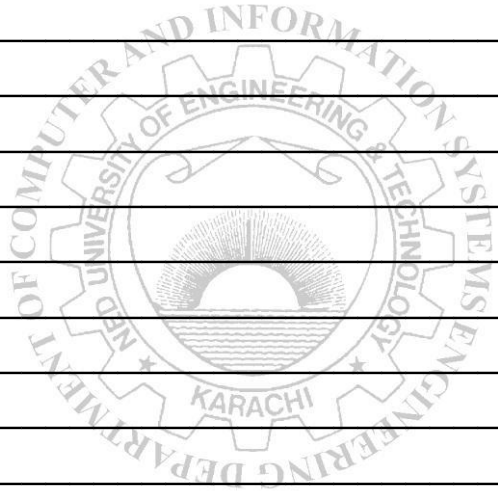
_____

_____

_____

_____

2. Define a frame for real world object ―Tea Cup‖, set its default attributes in your program. Display it to user and take input for changing default values.



3. Write a program, which illustrates the use of attribute chaining. Use the scenario of a shopping mall which is located at a particular place in a city of a certain country. The shopping mall contains a number of different shops each distinguished with a shop number.

4. Define a frame hierarchy for ―memory chips‖ that we use in computers, assign some default values and make use of positive and negative inheritance.

5. Define an action —Pc-purchase ‖ , make decision on the basis of parameters processor, RAM type, RAM capacity and hard disk capacity, and invoke it through console.

Lab Session 11

Defining questions and exploring the concept of ruleset in Flex.

Questions and Answers

Most expert system applications will involve some communication with the user. In *flex*, this is achieved by invoking pre-defined questions. These questions may involve making a selection from a menu, typing information at a keyboard, or indeed any set of operations which require a reaction by the user.

Defining Questions

Questions are defined within a KSL program by use of the keyword **`question`**. The main part of each question definition states how to obtain an answer, and what form it should take. *Flex* provides certain built-in constructs for obtaining answers, including single and multiple choice menus and data input screens. More sophisticated user interactions can be defined using the GUI facilities of the underlying Prolog system and can then easily integrated with the *flex* question and answer sub-system using relations or actions. Once a question is defined, it may be invoked using the KSL directives **`ask`** or **`answer`**, described later.

Menu Selection

In a menu selection format question, the user is presented with a collection of options, and is offered the choice of making either a single selection or multiple selections. The attraction of this is that the values obtained are implicitly validated, since they come from a fixed set of alternatives which have either been hard-wired into the question or programmatically generated.

Consider the problem of putting together a menu for some meal. The user is allowed to choose various combinations of dishes to make up that meal. For the main course, there is a straightforward choice between various meat dishes and fish dishes, only *one* of which can be selected.

The KSL for describing this question would be as follows:

```
question main_course
     Please select a dish for your main course; choose
one of steak,'lamb chops',trout,'dover sole' .
```

Whenever the question main_course is asked, the user is presented with a menu containing the items steak, 'lamb chops', trout and 'dover sole'. One, and only one, item can be selected from this menu.

Rather than state the items explicitly within the definition of the question, we can collect the items together and store them within a **group**. We can then refer to this name within the question.

```
group main_courses
   steak, 'lamb chops', trout, 'dover sole' .
question main_course
   Please select a dish for your main course ;
   choose one of main_courses .
```

Now, to accompany the main course there are various vegetables which can be selected in any

configuration. Using the KSL keywords **choose some of** we can allow the user to select any number of vegetables.

```
question vegetables
   Please select vegetables to accompany your main course ;
      choose some of potatoes, leeks, carrots, peas .
```

Storing Answers

Whenever a question is invoked, the ―answer‖ to the question is stored as the value of a global variable of the same name as the question. So, in the above two examples, the answers to the questions are stored in the global variables named **main_course** and vegetables respectively.

To test the above two questions we may define an action as follows. (Here we use of the KSL construct **check that** to ―retrieve ‖ the value of a global variable.)

```
action get_main_course( Main, Vegs ) ; do
   ask main_course and
   ask vegetables and
   check that Main is main_course and
   check that Vegs is vegetables .
```

Then `get_main_course/2` may be called from a Prolog program, or a *flex* procedure, or run from the Prolog command line, e.g.

```
?- get_main_course( M, V ).
M = steak, Vegs = [leeks, carrots, peas]
```

Alternatively we may define an action which simply prints out the values.

```
action show_main_course ; do
   ask main_course and ask
   vegetables and write(
   main_course ) and nl and
   write( vegetables ) .

?- show_main_course .

steak
[leeks, carrots, peas]
```

Keyboard Input

The second pre-defined question mechanism is through single field keyboard **input**. The data entered can easily be constrained to be either a text item (name), a floating-point number, an integer, or a set of such items.

Examples:

```
question    name_of_applicant
   Please enter your name ;
   input name .

question height_of_applicant
```

```
   Please enter your height (in metres) ;
   input number .

question address1_of_applicant Please
   enter your house number ; input
   integer .

question address5_of_applicant
   Please enter your city and post code ;
   input set .
```

Constrained Input

You can constrain the standard keyboard input to be something other than a name, number, integer or set of such objects, by nominating a Prolog program or *flex* relation to be used to validate the answer. This is indicated by the keywords **such that**.

```
question yes_or_no
   Please answer yes or no ;
   input K such that yes_or_no_answer( K ) .

relation yes_or_no_answer( yes ) .
relation yes_or_no_answer( no ) .
```

This will present a standard dialog (which will depend on your implementation of *flex*), and the user's response must satisfy the yes_or_no_answer relationship.

Customized Input

The range of standard questions provided will inevitably not cover all possible situations. For this reason, *flex* allows customized questions in which the programmer can specify both how to obtain an answer, and what form that answer should take. The onus is totally on the

programmer topresent the question to the user (for example create a dialog) and to return the appropriate answer. This is indicated by the KSL keyword **answer**.

```
question my_question
answer is K such that ask_my_question( K ) .
```

In this case no predefined dialog will be presented, but a call will be made to `ask_my_question/1`: this may be defined as a *flex* action, a *flex* relation or as a Prolog predicate. It should ask the question, creating any necessary dialogs, and return a value for the variable K.

Default Questions

When developing an application, it is often useful to delay the exact implementation of questions until some later stage. During this development process, *flex* allows you to declare a default question which is used in the absence of a specific definition. The name of the default question is catchall.

```
question catchall
      Please enter data ;
      input name .
```

Whenever a question is asked for which there is no definition, the catchall definition is used instead.

In this case, the default answer will be of type name.

Explaining Questions

In addition to the form of a question, you can optionally attach an explanation to any of the standard question types (as you can with rules) using a **because** clause.

The explanation itself can either be some canned text to be displayed, or it can be the name of a file to be browsed over. The explanations are presented whenever the end-user requests them (usually there is an *Explain* button in the built-in question dialogs).

```
question main_course
   Please select a dish for your main course ;
   choose from steak, 'lamb chops', trout, 'dover sole' ;
   because The main course is an integral part of a meal .

question headache
   Have you got a headache ? ;
   answer is K such that yes_or_no_answer( K ) ;
   browse file medical_symptoms .
```

Whenever there is a request to explain the headache question, the user begins browsing the file `medical_symptoms` starting at the headache topic. If explanations are attached to customized questions, then the onus is on the programmer to reflect any explanations to the end-user.

(Note that the following example will not work in MacProlog since there is no support for byte-level keyboard input.)

Example:

```
question headache answer
   is K such that
         write( 'Have you got a headache ?' ) and
         yes_no_question( K, medical_symptoms ) .

action yes_no_question( K, File ) ;
   do write( 'Please type Y or N or ESC' )
      and     repeat
      and     get0( Byte )
      and     yes_or_no_check( Byte, K, File )
      and ! .

relation yes_or_no_check( 89, yes, File ) .
relation yes_or_no_check( 121, yes, File ) .
relation yes_or_no_check( 78, no, File ) .
relation yes_or_no_check( 110, no, File ) .
relation yes_or_no_check( 26, _, File )
   if browse( File )
   and fail .
```

Invoking Questions

There are two underlying procedures, `ask/1` and `answer/2`, for invoking questions. These are reflected in the KSL as the directive.

```
ask <name of question>
```

and as the term

```
answer to <name of question>
```

respectively. These built-in procedures behave quite differently at run-time. Whenever there is a request to **ask** a question, that question is always asked immediately. However, a request for the **answer to** a question will only invoke that question if it has not previously been asked. If the behaviour of an application is such that the same question should be asked once, and only once, then use the **answer to** construct.

```
action decide_meal ;
      do ask main_course
      and ask vegetables .
```

In this case, using **as**k, the question dialogs will be displayed *every time* the `decide_meal` action is executed. The global values of `main_course` and vegetables will therefore change for each invocation of the `decide_meal` action.

```
rule prescription1
   if the answer to headache is yes
   and the answer to pregnant is no
   then prescribe( paracetamol ) .

rule prescription2
   if the answer to headache is yes and
   the answer to pregnant is yes then
   prescribe( nothing ) .
```

In this example, the actual questions headache and pregnant will only be asked if they haven‘t previously been asked. Once there is a value for each of headache and pregnant, this value will be used for the rest of the session. The questions headache and pregnant will therefore only be asked once.

Rules and Relations

In order to distinguish between rules which are intended to be used in a forward chaining manner from those which are to be used in a backward chaining manner, *flex* uses different rule formats and different KSL keywords. Forward chaining rules are indicated by the keyword **rule**; backward chaining rules are indicated by the keyword **relation**. Both formats fit the classical *if-then* style, but whereas backward chaining relations allow only for a single, positive conclusion in the *then* part, there is no such restriction in forward chaining rules which may have multiple conclusions, any of which may be either positive or negative in nature. Forward chaining rules often contain an action as part of their conclusion. This action usually updates various data (slot) values, which means that different rules will or will not fire next time round the forward chaining cycle.

Backward chaining, on the other hand, generally seeks to establish a logical sequence of rules and facts in order to prove a clause or goal. This does not involve rule firing or executing actions, but is of a more deductive nature.

Weighting of Rules

In rule-based systems there are always choice points where one rule is preferred to another. Attaching weights to rules is an option which can assist in making those decisions. The weight of a rule reflects its relative importance with respect to the other rules in the system. Whenever two or more rules are simultaneously applicable, their relative weights can be compared to decide which one to use. Most

weighting systems will be static, with each rule being assigned a specific score. The more important the rule, the higher its score should be. So, in the example below, if we have plenty of beer in the fridge, and the weather is hot, and it's late, then we will always drink beer regardless of the order of the rules.

```
rule drink_tea
     if the hour is late then
     drink_a_cup_of_tea score
     5 .

rule drink_beer

     if the fridge contains some beer
     and the weather is hot
     then drink_a_can_of_beer
     score 10 .
```

In addition, *flex* allows for dynamic weighting systems whereby the score attached to a rule is not fixed when the rule is defined, but is dependent upon some changing information.

For example, suppose we have a rule-based system for controlling the flow of materials between different storage vessels in a chemical plant, which contains a rule `empty_master_into_slave`. It is not easy to give an exact score for this rule which is accurate in all circumstances. At certain times this may be the most appropriate rule, but at other times there may be another rule, say emptying the master vessel completely, that is more important. To reflect this dependence upon the current circumstances, we can give the rule a dynamic score.

```
rule empty_master_into_slave
   if the master is not empty
   and the slave`s contents > the master`s spare_capacity

then fill_from( master, slave )
score master`s contents plus slave`s contents .
```

As the contents of the two vessels change, so the weighting of this rule also changes. We could even give the system a sense of chance by using random-valued scores.

Attaching Explanations to Rules

The final part of a production rule in *flex* involves the attachment of an optional explanation to rules. This is used to explain why a rule was fired. The explanation itself can either be some canned text or a pointer to a disk file accessed at run-time through a *file browse*r. When explaining the rule either the canned text is displayed on screen or the user is allowed to browse through the file.

```
rule 'check status'
   if the applicant`s job is officer
   and the applicant`s age is greater than 65
   then ask_for_grading
   because The job grading affects the pension .
```

The above rule will cause the single line of text The `job grading affects the pension` to be displayed on the screen whenever an explanation is requested. The latter alternative allows for a more flexible explanation facility. Indeed, the only purpose of the rule-based system might be as an intelligent link into the file browsing system. Such applications include the reviewing of technical manuals, especially medical journals.

```
rule 'check status'
```

```
if the applicant`s job is principal
and the applicant`s age is greater than 65
then check_principal_function

browse file status .
```

The Forward Chaining Engine

The forward chaining engine is implemented as a Prolog program. The emphasis of the implementation is placed upon simplicity coupled with great flexibility, rather than extracting the highest possible performance.

Ruleset

Rules are grouped together into *ruleset*s, and forward chaining is started using the KSL keywords **invoke ruleset**. This provides a construct for forming rules into stratified rule-bases and governing the forward-chaining engine in terms of rule selection method and agenda updating.

For example, to move a piece in a game we might have a ruleset called make_move containing rules for possible moves.

```
ruleset make_move
      contains corner_move, edge_move, centre_move .

action move ;
      invoke ruleset make_move .
```

The Rule Agenda

The rule agenda determines the rules currently available to the inference engine during forward chaining (initially specified by a ruleset). It might include all of the rules in the system, or only a subset. The rule agenda may contain duplicates, and there are no restrictions whatsoever on which names can go into the list. After a rule has been selected and fired, the rule agenda may be updated, and it is this revised agenda that the inference engine uses as the basis for the next cycle.

Setting The Initial Rule Agenda

The initial rule agenda is the set of rules from which the forward chaining engine makes its first selection. The initial rule agenda may be specified as containing all rules, a list of rules or a rule group. The only mandatory part of a given **ruleset** is the definition of the initial rule agenda.

```
ruleset everything contains
      all rules .
```

This KSL sentence sets the initial rule agenda to contain all the rules currently defined.

Rules Selection

A vital part of any engine, whether forward or backward chaining, is the method by which rules are selected. *flex* provides three built-in methods for selecting rules:

- First come first served
- Conflict resolution
- Conflict resolution with a threshold.

In addition, there is a facility for hooking in user-defined selection algorithms.

First Come First Served

The **`first come first served`** selection algorithm simply chooses the first rule in the agenda whose conditions (the **`if`** part) are all satisfied. First come first served is the default selection algorithm for the ruleset. For example;

```
ruleset mover
      contains push, pull, lift .

ruleset make_tea
      contains all rules ;
      select rule using first come first served .
```

The selection algorithm used in both of these examples will be first come first served.

Conflict Resolution

Conflict resolution is a more sophisticated and computationally expensive selection scheme whereby the "best" rule is always selected. A conflict occurs when more than one rule can be fired (i.e. the **`if`** conditions of more than one rule are satisfied). This conflict is then resolved by choosing the rule with the highest score. In the event of a tie the first is chosen. For Example;

```
ruleset mover.
      contains push, pull, lift
      select rule using conflict resolution .
```

Conflict Resolution with Threshold Values

A compromise between these two contrasting selection schemes, (first come first served and conflict resolution), is to introduce *threshold values* into the latter. This offers a conflict resolution scheme in principle, but which stops as soon as a candidate is found whose score is greater than the threshold value.

For example, if the agenda contained 1000 rules we might strike lucky and find that the 5 *th* rule can not only be fired, but also that its score is above the threshold value that has been set. Thus this rule will be fired with no further searching.

```
ruleset make_tea
      contains all rules ;
      select rule using conflict resolution
                  with threshold 6 .
```

Updating the Agenda

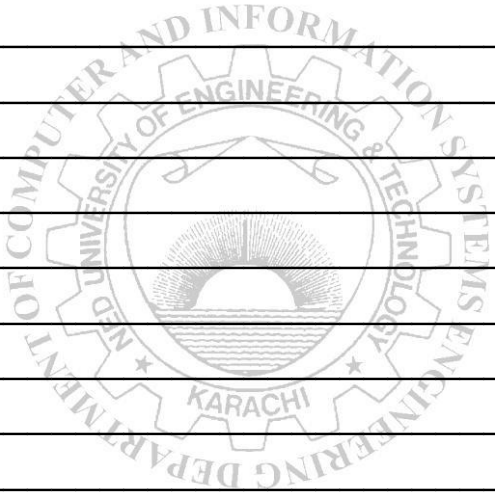
At the end of each cycle of the engine, the agenda can be updated according to the name of the rule which was fired. The default is to leave the set of rules exactly as it is, so that the rules are always considered in the same order every time.

EXERCISES

1. Write a program, which takes information a car (model, make, color, engine, registration number and country from where it is imported) as input, and displays it on screen.

2. Write a program for Menu selection in a restaurant. Make use of `Choose some of` clause.

3. Write a program suggesting user an appropriate dress for an occasion, making use of *ruleset*. Compose questions asking user about gender, weather condition, place, type of occasion, number of persons invited and relation with host/organizers etc. and suggest suitable outfit.


COMPUTER AND INFORMATION SYSTEMS ENGINEERING DEPARTMENT OF
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

4. Consider scenario of issuance of credit card to customers. Write a program which makes some decision whether credit card should be issued to a particular customer depending upon different conditions and codify it using ruleset.

Lab Session 12

Learning Data-Driven programming concepts

Data Driven Programming

The frame system of *flex* can be used for the representation of data and knowledge. In this chapter we describe data-driven programming, where rather than program the control-flow or logic of a system, procedures are attached to individual frames or groups of frames. These procedures lie dormant and are activated whenever there is a request to update, access or create an instance of the frame or slot to which they are attached. This concept of attached procedures, sometimes referred to as *active value*s, is also found in object-oriented programming.

Data-Driven Procedures

There are four types of data-driven procedures:

- Launches
- Watchdogs
- Constraints
- Demons

The following diagram shows when and where the various procedures are invoked.

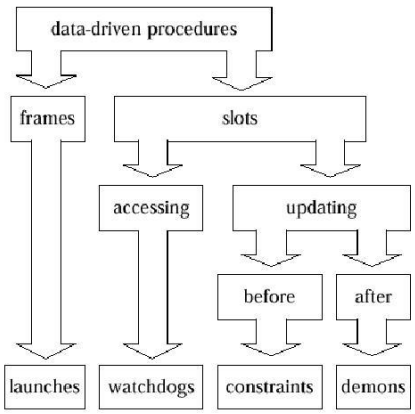


Figure 8.1

Launches

A **`launch`** procedure is activated whenever there is a request to create an instance of the frame to which it is attached.

A **`launch`** procedure has three main parts:

- context A test to see whether certain conditions hold.

- test A test to see whether certain conditions hold.

- action A series of commands to be performed.

The action will only be invoked if the test succeeds. The launch is invoked *after* the instance has been created.

Example:

Suppose we have a frame system representing a company's personnel structure, and that a new employee dave is to be added. The launch will automatically be invoked whenever a new instance of employee is requested, and provided the conditions hold, the actions will be performed.

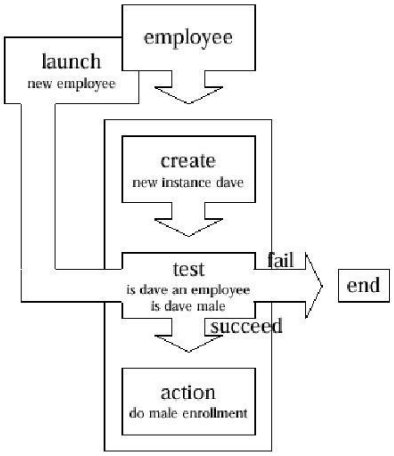


Figure 8.2

The launch procedure in this example, attached to the employee frame, is set up to collect the personal details about new male employees.
The KSL code for this example would be written as follows:

```
frame employee
   default sex is male .

launch new_employee
   when Person is a new employee
   and sex of Person is male
   then male_enrolment_questions(Person) .

instance dave is an employee .
```

The `male_enrolment_questions` will be defined elsewhere.

Constraining the Values of Slots

A **`constraint`** is attached to an individual slot, and is designed to constrain the contents of a slot to valid values. It is activated whenever the *current* value of that slot is updated, and the activation occurs immediately *before* the update.

A **`constraint`** has three main parts:

- context A test to see whether certain conditions hold.
- check A test to see whether the update is valid.

- error A series of commands to be performed for an invalid update.

The check will only be made if the context holds. If the check is successful then the update is allowed, otherwise the error commands are invoked and the update is not allowed.

Example:

If we had a frame system representing instances of water containers, we could put a constraint on the contents slot of any jug, such that when the value for the particular slot is being updated, a test is performed making sure that the new value is less than the value of the jug`s capacity, thus ensuring the jug does not overflow!

In the example the constraint is activated if the contents attribute of any jug changes. The prospective new value for the slot is then tested to see if it is less than that jug's capacity. If the test succeeds the update is allowed. If the test fails the update is not allowed and a message is displayed.

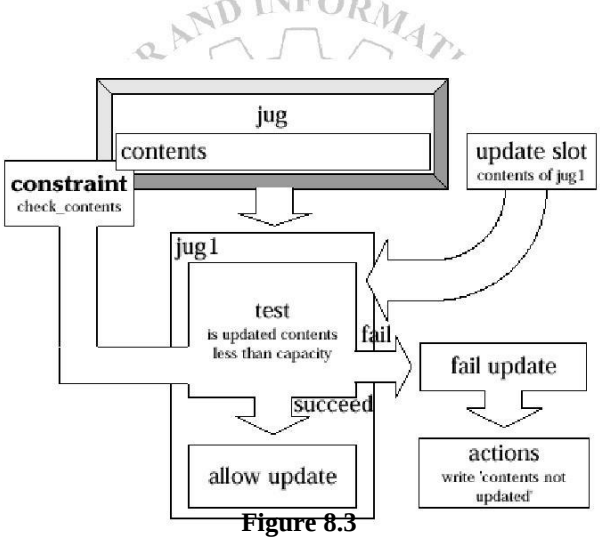


Figure 8.3

The code for this example could be written as follows:

```
frame jug
   default contents are 0 and
   default capacity is 7 .

instance jug1 is a jug .

constraint check_contents
   when the contents of Jug changes to X
   and Jug is some jug
   then check that number( X )
   and X =< Jug`s capacity
   otherwise write( 'contents not updated' )
   and nl .
```

Note the use of `Jug`, a local variable, which will unify with any frame or instance which has, in this case, a `contents` attribute.

Attaching Demons to Slot Updates

A `demon` is attached to an individual slot. It is activated whenever the *current* value of that slot is updated, and the activation occurs immediately *after* the update.

A **demon** has two main parts:

- context A test to see whether certain conditions hold.
- action A series of commands to be performed.

The slot is updated and then, given that the context holds, the actions will be invoked.

A **demon** can be tailored such that it fires only for given values and/or only under certain circumstances.

Example:

If we are modelling the action of kettles, we could attach a demon to the `temp` slot of any instance of the frame `kettle`.

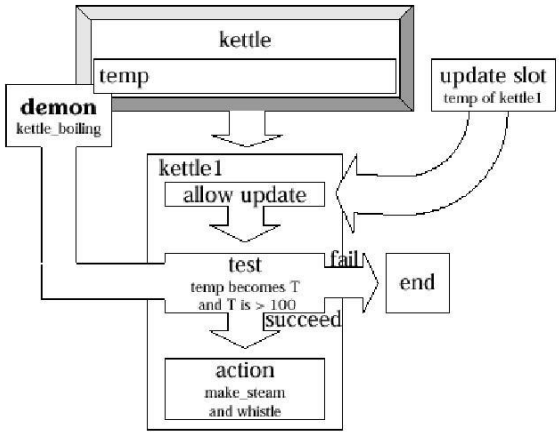


Figure 8.4

Whenever the temp slot is updated, a check will be made on the new value, such that, if it is greater than 100, then the actions make_steam and whistle are performed.

The code for this example could be written as follows:

```
frame kettle
   default temp is 0 .

instance kettle1 is a kettle .

demon kettle_boiling

   when the temp changes to T
   and T is greater than 100
   then make_steam
   and nl
   and whistle .
```

Restricting the Access to Slots

A **watchdog** is attached to an individual slot. It is activated whenever the *current* value of the slot is accessed.

A watchdog has three main parts:

- context A test to see whether certain conditions hold.
- check A test to see whether the access is valid.
- error A series of commands to be performed if access is invalid.

The check will only be made if the context holds. If the check is successful then the access is allowed. Otherwise, the error commands are invoked and the access is denied.

A watchdog can be used to check the access rights to an attribute of a frame. It is invoked whenever there is a request for the *current* value *(not* the default value) of that slot (attribute-frame pair).

Example:

In the example shown below, the watchdog is activated when the contents of a file are requested. A check on the user's classification is then made, and if the check succeeds the access is allowed. If the check fails the access is denied and a warning message is displayed.

The KSL for a similar example is shown below. In this case, only users with sufficiently high access priority may find out the balance in a bank account.

The current_user frame stores the access code of the current user, which is checked in the account_security watchdog.

```
frame bank_account
   default balance is 0.

frame current_user default
   name is '' and default
   access is 0.

watchdog account_security
   when the balance of Account is requested
   and Account is some bank_account
   then check current_user`s access is above 99 otherwise
   write( 'Balance access denied to user ' ) and
   write( current_user`s name ).
```

EXERCISES

1. Define a frame —Kettle ‖ and make use of launch procedure to create some instances.

2. Write a program which checks different conditions and permits/restricts user from accessing the attribute values. Use the scenario of ―Bank Account‖.

Lab Session 13

Learning MATLAB Fuzzy Logic Toolbox for the development of fuzzy logic based applications

Introduction

Fuzzy Logic Toolbox™ software is a collection of functions built on the MATLAB® technical computing environment. It provides tools for you to create and edit fuzzy inference systems within the framework of MATLAB. This toolbox relies heavily on graphical user interface (GUI) tools to help you accomplish your work, although you can work entirely from the command line if you prefer. In this lab session, we will use GUI for building fuzzy logic based application.

Fuzzy Inference Systems

Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. The mapping then provides a basis from which decisions can be made, or patterns discerned. The process of fuzzy inference involves ―*Membership Functions*‖, ―*Logical Operations*‖, and ―*If-Then Rules*‖. Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Because of its multidisciplinary nature, fuzzy inference systems are associated with a number of names, such as fuzzy-rule-based systems, fuzzy expert systems, fuzzy modeling, fuzzy associative memory, fuzzy logic controllers, and simply (and ambiguously) fuzzy systems.

You can implement two types of fuzzy inference systems in the toolbox: *Mamdani-type* and *Sugeno-type*. These two types of inference systems vary somewhat in the way outputs are determined.

Mamdani-type inference, as defined for the toolbox, expects the output membership functions to be fuzzy sets. After the aggregation process there is a fuzzy set for each output variable that needs defuzzification. It is possible, and in many cases much more efficient, to use a single spike as the output membership function rather than a distributed fuzzy set. This type of output is sometimes known as a *singleton* output membership function, and it can be thought of as a pre-defuzzified fuzzy set. It enhances the efficiency of the defuzzification process because it greatly simplifies the computation required by the more general Mamdani method, which finds the centroid of a two-dimensional function. Rather than integrating across the two-dimensional function to find the centroid, you use the weighted average of a few data points. Sugeno-type systems support this type of model. In general, Sugeno-type systems can be used to model any inference system in which the output membership functions are either linear or constant.

Overview of Fuzzy Inference Process

We‘ll start with a basic description of a two-input, one-output tipping problem (based on tipping practices in the United States).

The Basic Tipping Problem

Given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), and another number between 0 and 10 that represents the quality of the food at that restaurant (again, 10 is excellent), what should the tip be?

The starting point is to write down the three golden rules of tipping.

1. *If the service is poor or the food is rancid, then tip is cheap.*

2. *If the service is good, then tip is average.*
3. *If the service is excellent or the food is delicious, then tip is generous.*

Assume that an average tip is 15%, a generous tip is 25%, and a cheap tip is 5%.

Now that you know the rules and have an idea of what the output should look like, begin working with the GUI tools to construct a fuzzy inference system for this decision process.

The FIS Editor

The FIS Editor displays information about a fuzzy inference system. To open the FIS Editor, type the following command at the MATLAB prompt: `fuzzy`

The generic untitled FIS Editor opens, with one input labeled `input1`, and one output labeled `output1`.

In the given example, you construct a two-input, one output system. The two inputs are **`service`** and **`food`**. The one output is **`tip`**. To add a second input variable and change the variable names to reflect these designations:

1. Select **Edit > Add variable > Input**.
A second yellow box labeled **input2** appears.
2. Click the yellow box **input1**. This box is highlighted with a red outline.
3. Edit the **Name** field from input1 to service, and press **Enter**.
4. Click the yellow box **input2**. This box is highlighted with a red outline.
5. Edit the **Name** field from input2 to food, and press **Enter**.
6. Click the blue box **output1**.
7. Edit the **Name** field from output1 to tip, and press **Enter**.
8. Select **File > Export > To Workspace**.
9. Enter the **Workspace variable** name tipper, and click **OK**.

The diagram is updated to reflect the new names of the input and output variables. There is now a new variable in the workspace called tipper that contains all the information about this system. By saving to the workspace with a new name, you also rename the entire system. Your window looks something like the following diagram.

Leave the inference options in the lower left in their default positions for now. You have entered all the information you need for this particular GUI. Next, define the membership functions associated with each of the variables. To do this, open the Membership Function Editor. You can open the Membership Function Editor in one of three ways:

- Within the FIS Editor window, select **Edit > Membership Functions.**
- Within the FIS Editor window, double-click the blue icon called **tip**.
- At the command line, type `mfedit`.

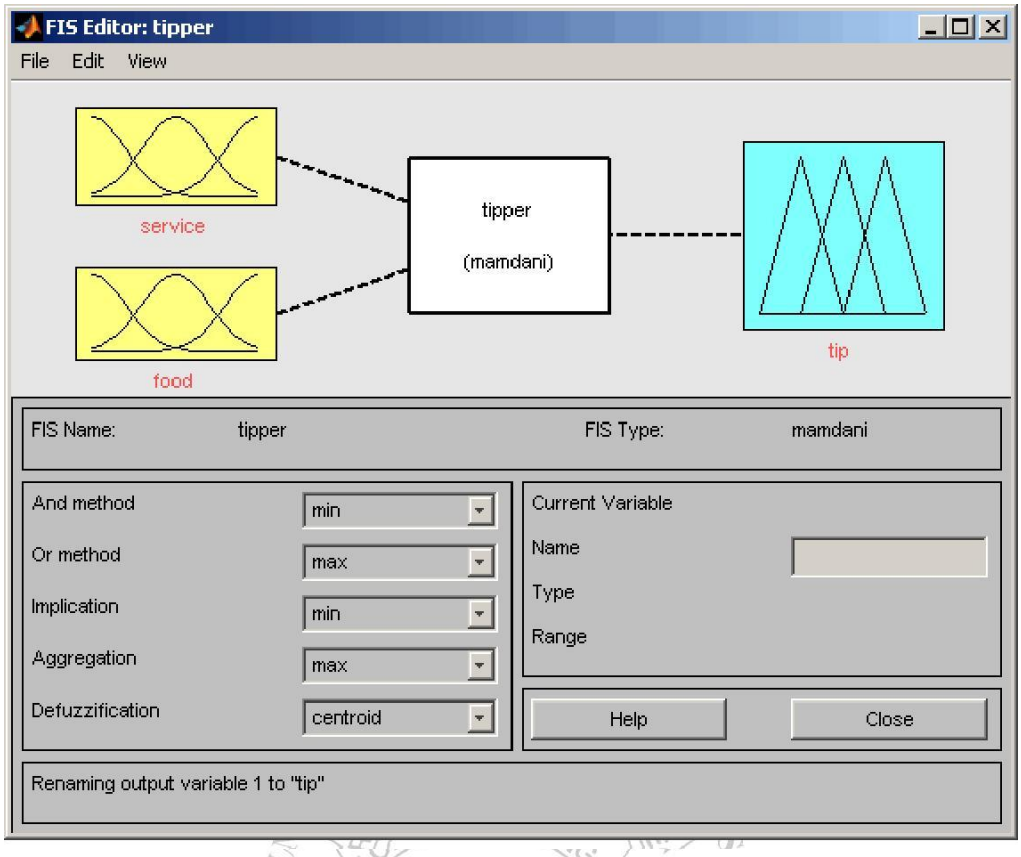


Figure 9.1

The Membership Function Editor

The Membership Function Editor is the tool that lets you display and edit all of the membership functions associated with all of the input and output variables for the entire fuzzy inference system. The Membership Function Editor shares some features with the FIS Editor, as shown in *Figure 2*. In fact, all of the five basic GUI tools have similar menu options, status lines, and **Help** and **Close** buttons.

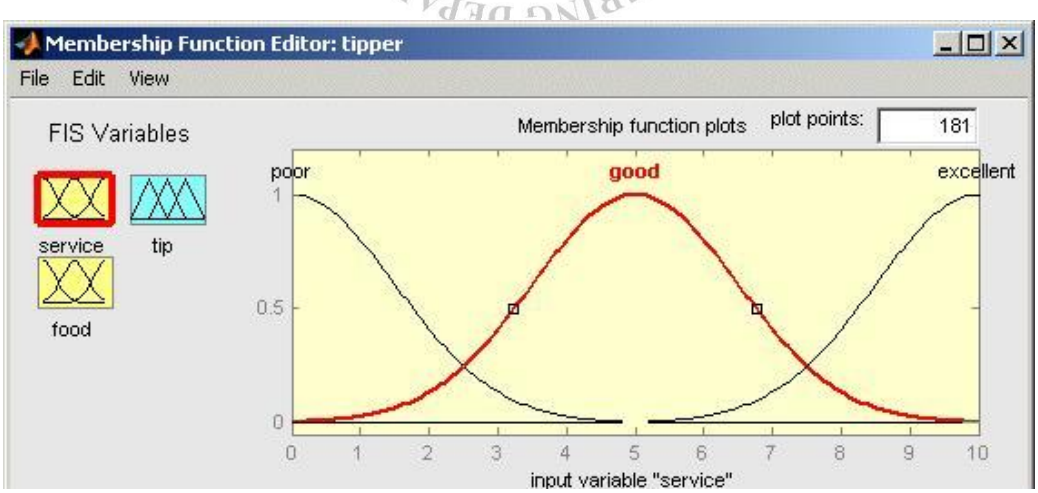


Figure 9.2

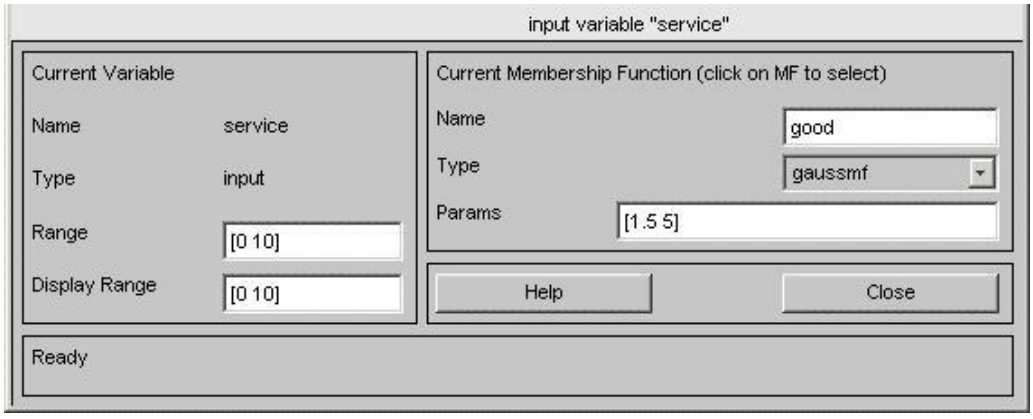


Figure 9.3

When you open the Membership Function Editor to work on a fuzzy inference system that does not already exist in the workspace, there is no membership function associated with the variables that you defined with the FIS Editor. On the upper-left side of the graph area in the

Membership Function Editor is a ―Variable Palette‖ that lets you set the membership functions for a given variable. To set up the membership functions associated with an input or an output variable for the FIS, select a FIS variable in this region by clicking it. Next select the **Edit** pull-down menu, and choose **Add MFs...** A new window appears which allows you to select both the membership function type and the number of membership functions associated with the selected variable.

In the lower-right corner of the window are the controls that let you change the name, type, and parameters (shape), of the membership function, after it is selected.

The membership functions from the current variable are displayed in the main graph. These membership functions can be manipulated in two ways. You can first use the mouse to select a particular membership function associated with a given variable quality, (such as poor, for the variable, service), and then drag the membership function from side to side.

The process of specifying the membership functions for the two input tipping example, tipper, is as follows:

1. Double-click the input variable service to open the Membership Function Editor.
2. In the Membership Function Editor, enter [0 10] in the **Range** and the **Display Range** fields.
3. Create membership functions for the input variable service.
   a. Select **Edit > Remove All MFs** to remove the default membership functions for the input variable service.
   b. Select **Edit > Add MFs.** to open the Membership Functions dialog box.
   c. In the Membership Functions dialog box, select gaussmf as the **MF Type**.
   d. Verify that 3 is selected as the **Number of MFs**.
   e. Click **OK** to add three Gaussian curves to the input variable service.
4. Rename the membership functions for the input variable service, and specify their parameters.
   a. Click on the curve named mf1 to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
      • In the **Name** field, enter poor.
      • In the **Params** field, enter [1.5 0].

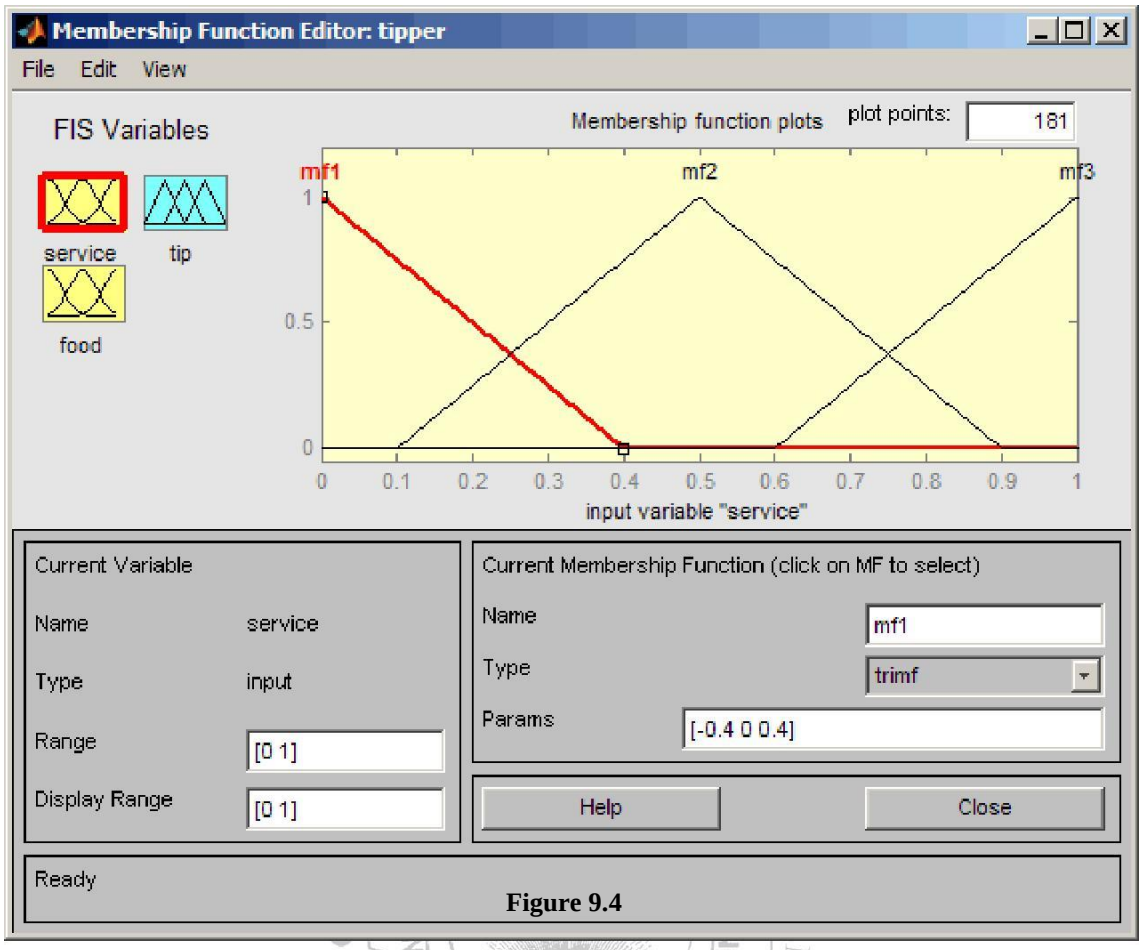


Figure 9.4

The two inputs of **Params** represent the standard deviation and center for the Gaussian curve. **Tip** to adjust the shape of the membership function, type in desired parameters.

b. Click on the curve named mf2 to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `good`.
- In the **Params** field, enter `[1.5 5]`.

c. Click on the curve named **mf3**, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `excellent`.
- In the **Params** field, enter `[1.5 10]`.

5. In the **FIS Variables** area, click the input variable `food` to select it.

6. Enter `[0 10]` in the **Range** and the **Display Range** fields.

7. Create the membership functions for the input variable `food`.

a. Select **Edit > Remove All MFs** to remove the default Membership Functions for the input variable `food`.

b. Select **Edit > Add MFs** to open the Membership Functions dialog box.

c. In the Membership Functions dialog box, select `trapmf` as the **MF Type**.

d. Select 2 in the **Number of MFs** drop-down list.

e. Click **OK** to add two trapezoidal curves to the input variable `food`.

8. Rename the membership functions for the input variable food, and specify their parameters:

a. In the **FIS Variables** area, click the input variable food to select it.

b. Click on the curve named **mf1**, and specify the following fields in the **Current**

Membership Function (click on MF to select) area:
- In the **Name** field, enter `rancid`.
- In the **Params** field, enter `[0 0 1 3]`.

c. Click on the curve named **mf2** to select it, and enter `delicious` in the **Name** field.
d. Reset the associated parameters if desired.

9. Click on the output variable `tip` to select it.

10. Enter `[0 30]` in the **Range** and the **Display Range** fields to cover the output range. The input ranges from 0 to 10, but the output is a `tip` between 5% and 25%.

11. Rename the default triangular membership functions for the output variable `tip`, and specify their parameters.

a. Click the curve named **mf1** to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `cheap`.
- In the **Params** field, enter `[0 5 10]`.

b. Click the curve named **mf2** to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `average`.
- In the **Params** field, enter `[10 15 20]`.

c. Click the curve named **mf3** to select it, and specify the following:
- In the **Name** field, enter `generous`.
- In the **Params** field, enter `[20 25 30]`.

Now that the variables have been named and the membership functions have appropriate shapes and names, you can enter the rules. To call up the Rule Editor, go to the **Edit** menu and select **Rules**, or type `ruleedit` at the command line.

Constructing rules using the graphical Rule Editor interface is fairly self evident. Based on the descriptions of the input and output variables defined with the FIS Editor, the Rule Editor allows you to construct the rule statements automatically, From the GUI, you can create rules by selecting an item in each input and output variable box, selecting one **Connection** item, and clicking **Add Rule**. You can choose none as one of the variable qualities to exclude that variable from a given rule and choose not under any variable name to negate the associated quality.

- Delete a rule by selecting the rule and clicking **Delete Rule**.
- Edit a rule by changing the selection in the variable box and clicking **Change Rule**.
- Specify weight to a rule by typing in a desired number between 0 and 1 in **Weight**. If you do not specify the weight, it is assumed to be unity (1).

To insert the first rule in the Rule Editor, select the following:
- `poor` under the variable **service**
- `rancid` under the variable **food**
- The **or** radio button, in the **Connection** block
- `cheap`, under the output variable, **tip**.

Then, click **Add rule**.

The resulting rule is
1. If (service is poor) or (food is rancid) then (tip is cheap) (1)

The numbers in the parentheses represent weights.

Follow a similar procedure to insert the second and third rules in the Rule Editor to get:

1. *If (service is poor) or (food is rancid) then (tip is cheap) (1)*
2. *If (service is good) then (tip is average) (1)*
3. *If (service is excellent) or (food is delicious) then (tip is generous) (1)*

At this point, the fuzzy inference system has been completely defined, in that the variables, membership functions, and the rules necessary to calculate tips are in place. Now, look at the fuzzy inference diagram presented at the end of the previous section and verify that everything is behaving the way you think it should. You can use the Rule Viewer, the next of the GUI tools we'll look at. From the **View** menu, select **Rules**.

The Rule Viewer

The Rule Viewer displays a roadmap of the whole fuzzy inference process. It is based on the fuzzy inference diagram described in the previous section. You see a single figure window *(Figure 5)* with 10 plots nested in it. The three plots across the top of the figure represent the antecedent and consequent of the first rule. Each rule is a row of plots, and each column is a variable. The rule numbers are displayed on the left of each row. You can click on a rule number to view the rule in the status line.

- The first two columns of plots (the six yellow plots) show the membership functions referenced by the antecedent, or the if-part of each rule.
- The third column of plots (the three blue plots) shows the membership functions referenced by the consequent, or the then-part of each rule.

Notice that under **food**, there is a plot which is blank. This corresponds to the characterization of none for the variable **food** in the second rule.

- The fourth plot in the third column of plots represents the aggregate weighted decision for the given inference system. This decision will depend on the input values for the system. The **defuzzified** output is displayed as a bold vertical line on this plot. The variables and their current values are displayed on top of the columns. In the lower left, there is a text field **Input** in which you can enter specific input values. For the two-input system, you will enter an input vector, `[9 8]`, for example, and then press **Enter**. You can also adjust these input values by clicking on any of the three plots for each input. This will move the red index line horizontally, to the point where you have clicked. Alternatively, you can also click and drag this line in order to change the input values. When you release the line, (or after manually specifying the input), a new calculation is performed, and you can see the whole fuzzy inference process take place.

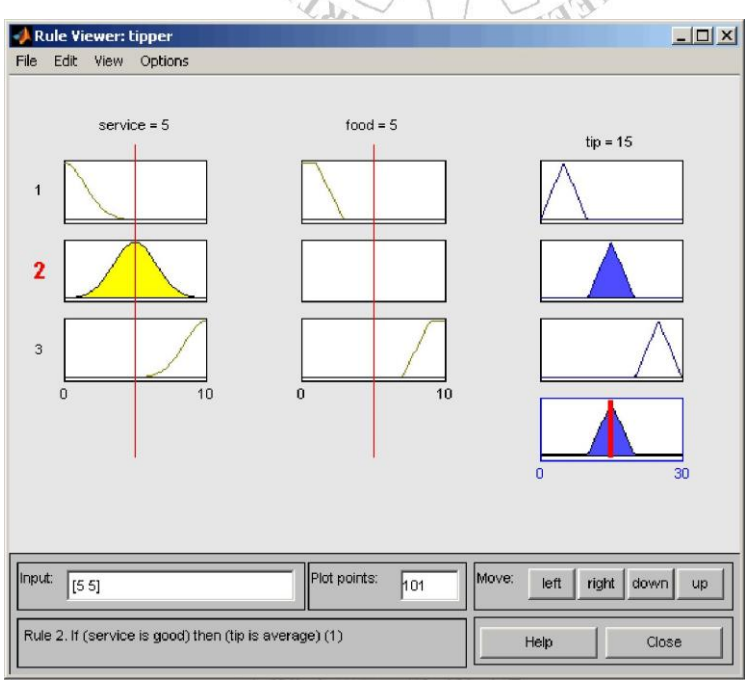


Figure 9.5

- Where the index line representing service crosses the membership function line —`service is poor`‖ in the upper-left plot determines the degree to which rule one is activated.
- A yellow patch of color under the actual membership function curve is used to make the fuzzy membership value visually apparent. Each of the characterizations of each of the variables is specified with respect to the input index line in this manner. If you follow rule 1 across the top of the diagram, you can see the consequent —`tip is cheap`‖ has been truncated to exactly the same degree as the (composite) antecedent—this is the implication process in action. The aggregation occurs down the third column, and the resultant aggregate plot is shown in the single plot appearing in the lower right corner of the plot field. The **defuzzified** output value is shown by the thick line passing through the aggregate fuzzy set. You can shift the plots using **left**, **right**, **down**, and **up**. The menu items allow you to save, open, or edit a fuzzy system using any of the five basic GUI tools.

The Rule Viewer allows you to interpret the entire fuzzy inference process at once. The Rule Viewer also shows how the shape of certain membership functions influences the overall result. Because it plots every part of every rule, it can become unwieldy for particularly large systems, but, for a relatively small number of inputs and outputs, it performs well (depending on how much screen space you devote to it) with up to 30 rules and as many as 6 or 7 variables.

When you save a fuzzy system to a file, you are saving an ASCII text FIS file representation of that system with the file suffix **.fis**. This text file can be edited and modified and is simple to understand. When you save your fuzzy system to the MATLAB workspace, you are creating a variable (whose name you choose) that acts as a MATLAB structure for the FIS system. FIS files and FIS structures represent the same system.

If you do not save your FIS to a file, but only save it to the MATLAB workspace, you cannot recover it for use in a new MATLAB session.

EXERCISES

1) Implement the following rule set in matlab, to control the mechanism of a crane.

   - IF Distance is far AND Angle is zero THEN apply pos_medium Power
   - IF Distance is far AND Angle is neg_small THEN apply pos_high Power
   - IF Distance is medium AND Angle is neg_small THEN apply pos_high Power
   - IF Distance is medium AND Angle is neg_big THEN apply pos_medium Power
   - IF Distance is close AND Angle is pos_small THEN apply neg_medium Power
   - IF Distance is close AND Angle is neg_small THEN apply pos_medium Power
   - IF Distance is close AND Angle is zero THEN apply zero Power
   - IF Distance is zero AND Angle is zero THEN apply zero Power
   - IF Distance is zero AND Angle is pos_small THEN apply neg_medium Power
   - Develop a fuzzy logic based application of your choice.

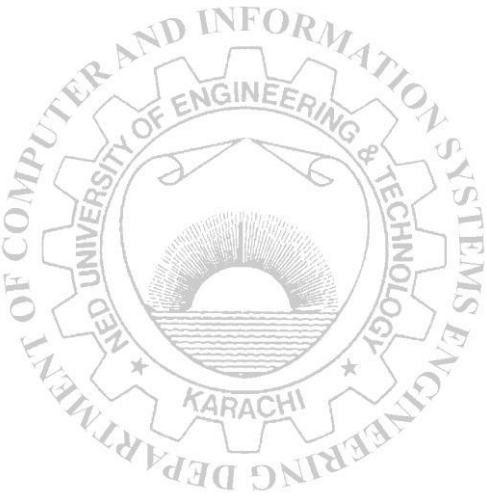
2) Attach the screenshots of FIS editor, membership functions of all input & output parameters, rule editor and rule viewer.

3) Implement the following rule set in matlab, to control the mechanism of a fuzzy logic based washing machine.

   - If clothe material is soft and status is clean then apply low power for less cycle time.
   - If clothe material is soft and status is dirty then apply low power for long cycle time.
   - If clothe material is medium and status is dirty then apply medium power for long cycle time.

- If clothe material is hard and status is clean then apply medium power for long cycle time.
- If clothe material is hard and status is dirty then apply high power for long cycle time.

4) Attach the screenshots of FIS editor, membership functions of all input & output parameters, rule editor and rule viewer.



Lab Session 14

Complex Engineering Activity: Design and implement of a Machine Learning application for a realtime problem

Course Code: CS 412, Course Title: Artificial Intelligence
CLO: CLO 2, Taxonomy Level: C3

Problem Statement

Design of an Artificial Neural Network for Real-Time Application

All those students of CIS who have enrolled in Artificial Intelligence course are required to design a project in the semester period. This project is about designing an Artificial Neural Network (ANN) for any valid real time application. By working on this project students will learn how to address a complex problem and how to implement ANN as a solution for some real time problem. Students are allowed to implement classification problem, prediction system, time-series analysis, image processing algorithm or any benchmark problem of ANN on some real time data. *Design a system/ algorithm for an Artificial Neural Network which addresses some Real-time Application.*

Motivation

ANNs are the backbone of machine learning. In this era of AI based revolution in the field of Computer Engineering, students must be able to implement ANNs/AI algorithm on real-time. This project will help students to understand the translation of ANN models into algorithmic details. This will provide a chance for students to apply learning algorithms in their project and directly measure the tune-ability (weight modification to global minima of error value) of their system.

EXERCISE

1. Explain you project in abstract form

_____

_____

_____

_____

_____

_____

_____

2. Highlight the major challenges in your project.

_____

_____

_____

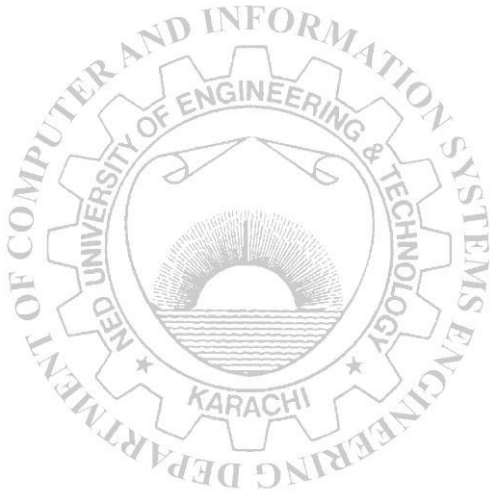
_____

_____

_____

_____

3. Attach the printout of your project code and outputs.



Course Code: CS 412, Course Title: Artificial Intelligence
Evaluation Rubric
CLO: CLO 2, Taxonomy Level: C3

CRITERIA AND SCALES			
Criterion 1: To what extent has the student has organized this project?			
0	1-2	3-4	
The project has been handled in a haphazard manner	The project has been partially organized	The project has been well organized	
Criterion 2: Is the student at ease with replying project related queries?			
0	1-2	3-4	
The student is not confident in his/her replies	The student is confident to some extent in his/her replies	The student is confident in his/her replies	
Criterion 3: How well has the student designed the project?			
0	1-2	3-4	
Student has no idea about the project	Student has moderate idea about the project	Student has complete idea about the project	
Criterion 4: Has the student been able to achieve the desired outputs?			
0	1-2	3-4	
The task is incomplete, no outputs have been achieved	Task has partially been completed on time, the outputs are erroneous	Task has been completed on time, desired outputs have been achieved	
Criterion 5: How would you grade the interaction of the student with lab resources (lab personnel, participant students, equipment)?			
0	1-2	3-4	
The student took no notice of the lab resources	The student was aware of lab resources for a short period of time but was mostly unconcerned	The student effectively interacted with the lab resources	
Criterion 6: What is the student's level of confidence with the Simulation Tool Interface?			
0	1-2	3-4	5
The student is unfamiliar with the tool	The student is familiar with the visible features of the tool	The student is familiar with the unexposed features of the tool	The student is proficient with the tool

Course Code: CS 412, Course Title: Artificial Intelligence
Evaluation Rubric
CLO: CLO 2, Taxonomy Level: C3

CRITERIA AND SCALES			
Criterion 1: To what extent has the student has organized this project?			
0	1-2	3-4	
The project has been handled in a haphazard manner	The project has been partially organized	The project has been well organized	
Criterion 2: Is the student at ease with replying project related queries?			
0	1-2	3-4	
The student is not confident in his/her replies	The student is confident to some extent in his/her replies	The student is confident in his/her replies	
Criterion 3: How well has the student designed the project?			
0	1-2	3-4	
Student has no idea about the project	Student has moderate idea about the project	Student has complete idea about the project	
Criterion 4: Has the student been able to achieve the desired outputs?			
0	1-2	3-4	
The task is incomplete, no outputs have been achieved	Task has partially been completed on time, the outputs are erroneous	Task has been completed on time, desired outputs have been achieved	
Criterion 5: How would you grade the interaction of the student with lab resources (lab personnel, participant students, equipment)?			
0	1-2	3-4	
The student took no notice of the lab resources	The student was aware of lab resources for a short period of time but was mostly unconcerned	The student effectively interacted with the lab resources	
Criterion 6: What is the student's level of confidence with the Simulation Tool Interface?			
0	1-2	3-4	5
The student is unfamiliar with the tool	The student is familiar with the visible features of the tool	The student is familiar with the unexposed features of the tool	The student is proficient with the tool