The Concept of Learning

- Learning is the ability to adapt to new surroundings and solve new problems.

- “Modification of a behavioural tendency by expertise.” (Webster 1984)

- “A learning machine, broadly defined is any device whose actions are influenced by past experiences.” (Nilsson 1965)

- “Any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population.” (Simon 1983)

- “An improvement in information processing ability that results from information processing activity.” (Tanimoto 1990)
Main types of learning

- Rote Learning (Memorisation)
- Taking Advice (Direct Instruction)
- Learning in Problem Solving
- Learning from Examples (Induction)
- Explanation-based Learning
- Discovery (Deduction)
- Analogy
- Neural Nets and Genetic Algorithms

Rote Learning (Memorisation)

- Simple storage of data – data caching.

- Requires:
  1. Organised storage of information, and
  2. Generalisation.

- Eg. Samuel’s checkers program – stores scores from minimax search

- no need to re-evaluate a particular sub-tree at a later stage.
**Learning by Taking Advice (Direct Instruction)**

- Involves receiving direct instructions on how to respond to certain situations.

- In a machine, this amounts to straightforward procedural programming.

- In situations where the instructions do not correspond to direct procedures (e.g. “Take control of the centre of the board” in Chess), an interpreter is required to translate the instructions to concrete execution steps.

**Learning in Problem Solving**

- Learning ways of problem solving **from own experience** – without an instructor/advisor.

- Does not involve an increase in knowledge, just the methods in using the knowledge.

- The Utility Problem: Learnt rules are good in directing the problem solving process, **but** it incurs a cost (utility) because the problem solver needs to store and consult those rules.

- Can be partially overcome by a utility measure to keep track of how useful the learnt rules are, and deleting them when necessary.
Types of Learning in Problem Solving

1. Learning by Parameter Adjustment

   - Use outcomes to adjust the weights for factors in an evaluation function

   - Considerations:
     What are the initial weights?
     When does certain weights increase?
     When do they decrease?

2. Learning with Macro-operators

   - Rote Learning a sequence of operations found to be useful during problem solving.

3. Learning by Chunking

   - Rote learning in the context of a Production System

   - Rules which are useful and always fired together are chunked to form one large production.
Learning from Examples (Induction)

- Classification: the process of assigning, to a particular input, the name of a class to which it belongs.

- Classify by looking at many different examples of a class, and generalising the common features.

- Must define class structure before classification process begins.

Explanation Based Learning

- To extract the concept behind the information within one example, and generalize to other instances.

- Requires domain-specific knowledge.

- In general, the inputs to EBL programs are:
  1. A Training Example
  2. A Goal Concept
  3. An Operationally Criterion
  4. A Domain Theory (or Knowledge Base)
DISCOVERY (DEDUCTION)

- Much like Learning in Problem Solving, this involves gleaming information without the use of a teacher.

- Focuses more on extracting knowledge, rather than strategies/operations in problem solving.

ANALOGY

- Eg: “Last month, the stock market was like a roller-coaster.”

- Transformational Analogy

  - Transform solutions previously found to new solutions.

  ![Diagram of Transformational Analogy]

  New Problem  \(\rightarrow\)  Previously solved

  Solution to new problem  \(\rightarrow\)  Solution to old problem

  Transform

Eg. Transforming a PASCAL program to a C program.
- Derivational Analogy

  - Use methods in previously solved problems to derive methods of solving new problems.

![Diagram of Derivational Analogy]

Eg. Learning ways of deconstructing lists in PROLOG by studying example clauses.

**NEURAL NETS AND GENETIC LEARNING**

- Learning by iterative improvement: start with an initial (possibly random) solution, then improve on the solution step-by-step.
- Genetic Learning: based on *evolution* and *natural selection*.
- ‘evolve’ new solutions from old ones, then ‘selection’ the new solutions which are good.
How Brain Works

- A neural network can be defined as a model of reasoning based on the human brain.

- The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called neurons.

- The human brain incorporates nearly 10 billion neurons and 60 trillion connections, synapses, between them.

- Our brain can be considered as a highly complex, non-linear and parallel information-processing system.

- Information is stored and processed in a neural network simultaneously throughout the whole network, rather than at specific locations.

- Learning is a fundamental and essential characteristic of biological neural networks.

- The ease with which they can learn led to attempts to emulate a biological neural network in a computer.
Artificial Neural Networks

- An Artificial Neural Network (ANN) consists of a number of very simple processors, also called *neurons*, which are analogous to the biological neuron in the brain.

- The neurons are connected by weighted links passing signals from one neuron to another.

- The output signal is transmitted through the neuron’s outgoing connection.

- The outgoing connection splits into a number of branches that transmit the same signal. The outgoing branches terminate at the incoming connections of other neurons in the network.

- Architecture of a typical artificial neural network
- The neuron as a simple computing element

<table>
<thead>
<tr>
<th>Input Signals</th>
<th>Weights</th>
<th>Output Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$w_1$</td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>$w_2$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$x_n$</td>
<td>$w_n$</td>
<td></td>
</tr>
</tbody>
</table>

- The neuron computes the weighted sum of the input signals and compares the results with a threshold value, $\theta$.

- If the net input is less than the threshold, the neuron output is $-1$. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value $+1$.

- The neuron uses the following *transfer or activation function*:

  $$X = \sum_{i=1}^{n} x_i w_i$$
  
  $$Y = \begin{cases} 
  +1, & \text{if } X \geq \theta \\
  -1, & \text{if } X < \theta 
  \end{cases}$$

- This type of activation function is called a sign function.
• Activation functions of a neuron

\[
\begin{align*}
Y_{\text{step}} &= \begin{cases} 
1, & \text{if } X \geq 0 \\
0, & \text{if } X < 0 
\end{cases} \\
Y_{\text{sign}} &= \begin{cases} 
+1, & \text{if } X \geq 0 \\
-1, & \text{if } X < 0 
\end{cases} \\
Y_{\text{sigmoid}} &= \frac{1}{1+e^{-X}} \\
Y_{\text{linear}} &= X
\end{align*}
\]

• In 1958, Frank Rosenblatt introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron**.

• The perceptron is the simplest form of a neural network. It consists of a single neuron with adjustable synaptic weights and a **hard limiter**.

**Inputs**

- \(x_1\)
- \(x_2\)

**Linear Combiner**

- \(w_1\)
- \(w_2\)

\[\Sigma\]

**Hard Limiter**

\[\Theta\]

**Threshold**

**Output**

- \(Y\)
Perceptron

- The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to +1 if its input is positive and –1 if it is negative.

- The aim of the perceptron is to classify inputs, $x_1, x_2, \ldots, x_n$, into one of two classes, say $A_1$ and $A_2$.

- In the case of an elementary perceptron, the $n$-dimensional space is divided by a hyperplane into two decision regions. The hyperplane is defined by the *linearly separable* function:

$$
\sum_{i=1}^{n} x_i w_i - \theta = 0
$$

\[ (a) \text{ Two-input perceptron.} \quad (b) \text{ Three-input perceptron.} \]
The perceptron learn its classification tasks by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron.

The initial weights are randomly assigned, and then updated to obtain the output consistent with the training examples.

If at iteration $p$, the actual output is $Y(p)$ and the desired output is $Y_d(p)$, then the error is given by:

$$e(p) = Y_d(p) - Y(p)$$

where $p = 1, 2, 3, \ldots$

Iteration $p$ here refers to $p$th training example presented to the perceptron.

If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

The perceptron learning rule was first proposed by Rosenblatt in 1960:

$$w_i(p + 1) = w_i(p) + \alpha \cdot x_i(p) \cdot e(p)$$

where $p = 1, 2, 3, \ldots$

$\alpha$ is the learning rate, a positive constant less than unity.
Perceptron’s Training Algorithm

- **Step 1: Initialisation**
  - Set initial weights $w_1, w_2, \ldots, w_n$ and threshold $\theta$ to random numbers in the range $[-0.5, 0.5]$.
  - If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

- **Step 2: Activation**
  - Activate the perceptron by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired output $Y_d(p)$.
  - Calculate the actual output at iteration $p = 1$
    $$Y(p) = \text{step} \left[ \sum_{i=1}^{n} x_i(p) w_i(p) - \theta \right]$$
    where $n$ is the number of the perceptron inputs, and $\text{step}$ is a step activation function.
- **Step 3: Weight training**
  
  - update the weights of the perceptron
    \[ w_i(p + 1) = w_i(p) + \Delta w_i(p) \]
    where \( w_i(p) \) is the weight correction at iteration \( p \).

  - The weight correction is computed by the delta rule:
    \[ \Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p) \]

- **Step 4: Iteration**
  
  - Increase iteration \( p \) by one, go back to *Step 2* and repeat the process until convergences.
- **Examples of perceptron learning: the logical operation AND**

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Inputs</th>
<th>Desired output $Y_d$</th>
<th>Initial weights $w_1$, $w_2$</th>
<th>Actual output $Y$</th>
<th>Error $e$</th>
<th>Final weights $w_1$, $w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0</td>
<td>0</td>
<td>0.3, -0.1</td>
<td>0</td>
<td>0</td>
<td>0.3, -0.1</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0</td>
<td>0.3, -0.1</td>
<td>0</td>
<td>0</td>
<td>0.3, -0.1</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>0</td>
<td>0.3, -0.1</td>
<td>1</td>
<td>-1</td>
<td>0.2, -0.1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1</td>
<td>0.2, -0.1</td>
<td>0</td>
<td>1</td>
<td>0.3, 0.0</td>
</tr>
<tr>
<td>2</td>
<td>0 0</td>
<td>0</td>
<td>0.3, 0.0</td>
<td>0</td>
<td>0</td>
<td>0.3, 0.0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0</td>
<td>0.3, 0.0</td>
<td>0</td>
<td>0</td>
<td>0.3, 0.0</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>0</td>
<td>0.3, 0.0</td>
<td>1</td>
<td>-1</td>
<td>0.2, 0.0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1</td>
<td>0.2, 0.0</td>
<td>1</td>
<td>0</td>
<td>0.2, 0.0</td>
</tr>
<tr>
<td>3</td>
<td>0 0</td>
<td>0</td>
<td>0.2, 0.0</td>
<td>0</td>
<td>0</td>
<td>0.2, 0.0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0</td>
<td>0.2, 0.0</td>
<td>0</td>
<td>0</td>
<td>0.2, 0.0</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>0</td>
<td>0.2, 0.0</td>
<td>1</td>
<td>-1</td>
<td>0.1, 0.0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1</td>
<td>0.1, 0.0</td>
<td>1</td>
<td>0</td>
<td>0.2, 0.1</td>
</tr>
<tr>
<td>4</td>
<td>0 0</td>
<td>0</td>
<td>0.2, 0.1</td>
<td>0</td>
<td>0</td>
<td>0.2, 0.1</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0</td>
<td>0.2, 0.1</td>
<td>0</td>
<td>0</td>
<td>0.2, 0.1</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>0</td>
<td>0.2, 0.1</td>
<td>1</td>
<td>-1</td>
<td>0.1, 0.1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1</td>
<td>0.1, 0.1</td>
<td>1</td>
<td>0</td>
<td>0.1, 0.1</td>
</tr>
<tr>
<td>5</td>
<td>0 0</td>
<td>0</td>
<td>0.1, 0.1</td>
<td>0</td>
<td>0</td>
<td>0.1, 0.1</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0</td>
<td>0.1, 0.1</td>
<td>0</td>
<td>0</td>
<td>0.1, 0.1</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>0</td>
<td>0.1, 0.1</td>
<td>0</td>
<td>0</td>
<td>0.1, 0.1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1</td>
<td>0.1, 0.1</td>
<td>1</td>
<td>0</td>
<td>0.1, 0.1</td>
</tr>
</tbody>
</table>

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$
Multilayer Neural Networks (MLNN)

- A multiplayer perceptron is a feedforward neural network with one or more hidden layers.

- The network consists of an *input layer* of source neurons, at least one *hidden layer* of computational neurons, and an *output layer* of computational neurons.

- The input signals are propagated in a forward direction on a layer-by-layer basis.

- Neurons in the hidden layer cannot be observed through input/output behaviour of the network.

- There is no obvious way to know what the desired output of the hidden layer should be.
Backpropagation Neural Network (BPNN)

- Learning in a multiplayer network proceeds the same way as for a perceptron

- A training set of input patterns is presented to the network

- The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error

- In BPNN, the learning algorithm has two phases

- First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer

- If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.
Training of Backpropagation Algorithm

- Step 1: Initialisation

  - Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range:

    \[
    \left( -\frac{2.4}{F_i}, \frac{2.4}{F_i} \right)
    \]

    where $F_i$ is the total number of inputs of neuron $i$ in the network. The weight initialisation is done on a neuron-by-neuron basis.
Step 2: Activation

- Activate the BPNN by applying inputs $x_1(p), x_2(p), ..., x_n(p)$ and desired outputs $y_{d,1}(p), y_{d,2}(p), ..., y_{d,n}(p)$.

- Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = \text{sigmoid} \left[ \sum_{i=1}^{n} x_i(p) \cdot w_{ij}(p) - \theta_j \right]$$

where $n$ is the number of inputs of neuron $j$ in the hidden layer, and sigmoid is the sigmoid activation function.

- Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = \text{sigmoid} \left[ \sum_{j=1}^{m} x_{jk}(p) \cdot w_{jk}(p) - \theta_k \right]$$

where $m$ is the number of inputs of neuron $k$ in the output layer.
- Step 3: Weight Training
  - Update the weights in the BPNN by propagating backward the errors associated with output neurons.

  - Calculate the error gradient for the neurons in the output layer:
    \[ \delta_k(p) = y_k(p) \cdot [1 - y_k(p)] \cdot e_k(p) \]
    where \[ e_k(p) = y_{d,k}(p) - y_k(p) \]
    Calculate the weight corrections:
    \[ \Delta w_{jk}(p) = \alpha \cdot y_j(p) \cdot \delta_k(p) \]
    Update the weights at the output neurons:
    \[ w_{jk}(p + 1) = w_{jk}(p) + \Delta w_{jk}(p) \]

  - Calculate the error gradient for the neurons in the hidden layer:
    \[ \delta_j(p) = y_j(p) \cdot [1 - y_j(p)] \cdot \sum_{k=1}^{l} \delta_k(p) \cdot w_{jk}(p) \]
    Calculate the weight corrections:
    \[ \Delta w_{ij}(p) = \alpha \cdot x_i(p) \cdot \delta_j(p) \]
    Update the weights at the hidden neurons:
    \[ w_{ij}(p + 1) = w_{ij}(p) + \Delta w_{ij}(p) \]

- Step 4: Iteration
• Increase iteration $p$ by one, go back to Step 2 and repeat the process until the selected error criterion is satisfied.