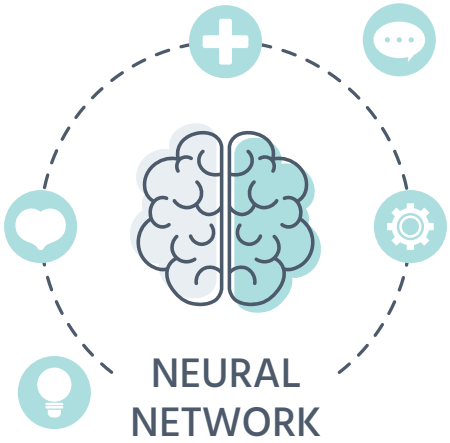
Cracking The Neural Computing



Unraveling the Neural Network Frontier: A Journey into Deep Learning

July 2023

Kanchon Gharami

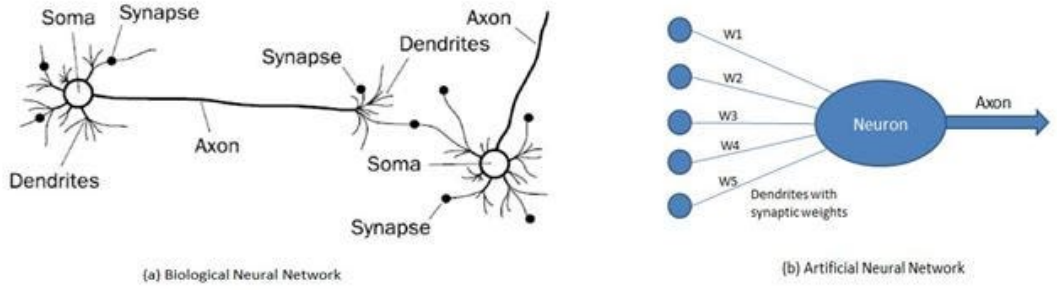
Contents

Chapter 1

Introduction

1.1 Learning in Biological Neurons

In biology, learning occurs through a process known as *synaptic plasticity*, where the strength or effectiveness of the synapses (connections between neurons) changes over time. This process is often governed by *Hebb's rule*, which is commonly summarized as "neurons that fire together, wire together". In other words, if a neuron A consistently helps to fire another neuron B, then the synapse from A to B is strengthened, as depicted in the figure below.



This is a simplified explanation, and the actual mechanisms of synaptic plasticity and learning in biological systems are far more complex and not yet fully understood.

1.2 Graceful Degradation

Graceful degradation refers to the capability of the network to maintain performance and continue functioning, albeit at a reduced efficiency, even when certain parts of the network are damaged or some neurons are removed. In real-world applications, this means that even if

some data is lost or corrupted, the overall performance of the network decreases slowly rather than failing entirely.

Key characteristics of graceful degradation include:

- **Fault Tolerance**: The network can handle faults or errors without completely failing.
- **Robustness**: The network maintains its performance even in the face of partial network failures.
- **Redundancy**: Multiple neurons often store the same information, so the loss of a few neurons does not result in the loss of that information.

1.3 Humans Brain and Computers

Table 1.1: Comparison between Human Brain and Machines

Aspect	Human Brain	Machine/Artificial Intelligence
Processing Speed	Comparatively slow (neurons fire at millisecond intervals)	Extremely fast, processing data at lightning speed
Parallel/Serial Computing	Highly capable of parallel processing	Primarily serial processing
Computing Speed	High computing speed due to parallel processing	Limited processing speed due to serial nature
Pattern Recognition	Excellent pattern recognition capability	Requires vast amounts of labeled data for training

1.4 Effective Coupling for Information Transfer

Effective coupling refers to the effective connection and communication between neurons in a neural network. It determines the efficiency of information transfer in the network. The strength and nature of these connections (synapses) can significantly influence the network's learning capability and its capacity to solve complex problems.

Key characteristics of effective coupling include:

- **Connection Strength**: Strong connections allow for more efficient transfer of information.
- **Connection Pattern**: The structure and pattern of connections can influence the network's learning ability and functionality.

- **Dynamic Adaptability**: In many neural networks, connections can adapt and change their strength based on the learning process (synaptic plasticity).

1.5 Naive Bayes Classifier

Algorithm 1: Naive Bayes Classifier

input : Training set $\mathcal{D}$, New instance x

output: Class label for x

Step 1: Calculate Prior Probabilities for all classes:

$$P(c_j) = \frac{|\{d \in \mathcal{D} : class(d) = c_j\}|}{|\mathcal{D}|}$$

Step 2: Calculate Likelihoods for all features given each class:

$$P(f_i|c_j) = \frac{|\{d \in \mathcal{D} : feature(d, i) = f_i \wedge class(d) = c_j\}|}{|\{d \in \mathcal{D} : class(d) = c_j\}|}$$

Step 3: Calculate Posterior Probabilities for all classes:

$$P(c_j|x) = P(c_j) \times \prod_i P(f_i|c_j)$$

Step 4: Return the class with maximum posterior probability:

$$c^* = \arg\max_{c_j} P(c_j|x)$$

Chapter 2

Pattern Recognition

2.1 Pattern

2.1.1 What Is Pattern?

A pattern in the context of neural computing refers to a regularity in the world or in a manmade design. It can also be thought of as a set of numerical or symbolic values that characterize a certain feature or object.

For instance, consider a sequence of numbers like 2, 4, 6, 8, 10. Here, the pattern can be described as "an increase of 2".

Here is an example of a pattern in a table:

Index	Value
1	2
2	4
3	6
4	8
5	10

Table 2.1: Example of a Numerical Pattern

The pattern in the above table is a sequence of even numbers where each number is increased by 2 from the previous one.

2.1.2 What Is Pattern Recognition?

Pattern recognition is a branch of machine learning that focuses on the detection and identification of patterns and regularities in data.

Steps of Pattern Recognition

Pattern recognition generally involves two key steps:

1. **Feature Extraction** Feature extraction is the process of transforming the raw data into a set of features, or "feature vectors", that can effectively represent the underlying patterns in a simpler way.
    - A measurement taken on the input pattern
2. **Classification** Classification is the task of assigning the object of interest to a predefined class or category based on its features.
    - Map these input features onto a classification state

It involves classifying input data into objects or classes based on key features, using either statistical(Numeric) or syntactic(Non-Numeric) approach.

- **Numeric Approach:** deterministic and statistical measures – made on the geometric pattern space
- **Non-Numeric Approach:** based on symbolic processing – i.e. – fuzzy set.

2.2 Feature Vectors

A feature vector is an n-dimensional vector of numerical features that represent some object.

These features are often measurements or descriptors that quantify the characteristics of the object in some meaningful way. For example, in image recognition, a feature vector might include values indicating color intensity, texture patterns, shapes present, etc.

2.3 Feature Space

Feature space refers to the n-dimensional space in which these feature vectors exist.

Each dimension represents a single feature, and the position of an object in this space is determined by its feature vector. The goal in pattern recognition and machine learning is often to find surfaces or boundaries in this space that separate objects of different classes or groups.

For instance, if we have a feature vector $F = [f_1, f_2, ..., f_n]$, its feature space is $\mathbb{R}^n$.

2.4 Discriminant Functions and Decision Boundaries

2.4.1 Discriminant Functions

Algebraic representation of a decision boundary is refer as discriminant functions.

In the context of pattern recognition, discriminant functions are used to determine to which category or class a new observation belongs. The decision is made based on a set of measurements, known as features, taken from the observation. Simply put, given an input (a feature vector), a discriminant function assigns it to one of several predefined categories.

2.4.2 Decision Boundaries

A decision boundary is a hypersurface that partitions the underlying input space into two or more regions. Each region corresponds to a class, and the decision boundary serves as the "line" that differentiates between these classes.

1. The simplest function that separate the two clusters is a straight line.
2. It represents a very widely used category of classifiers known as linear classifiers.

2.5 Methods of measurement based on distance

Distance Metrics

In the context of machine learning and pattern recognition, a distance metric, also known as a distance function, is a function that defines a distance between elements of a set. The output of this function, the distance, is a scalar measure of the dissimilarity or separation between two elements.

Common examples of distance metrics include Euclidean distance, Manhattan distance, and Hamming distance.

- **Euclidean Distance**: Also known as L^2 distance, it measures the "as-the-crow-flies" distance. Given two points $P = (p_1, p_2, ..., p_n)$ and $Q = (q_1, q_2, ..., q_n)$, the Euclidean distance is defined as:

$$d_{\text{euclidean}}(P, Q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

- **Manhattan Distance**: Also known as L^1 distance or city-block distance, it is the distance a taxi would drive in a city (made up of a grid). For points P and Q, it's defined as:

$$d_{\text{manhattan}}(P, Q) = |q_1 - p_1| + |q_2 - p_2| + \cdots + |q_n - p_n|$$

- **Hamming Distance**: Used for binary vectors, it is the number of bit flips required to convert one binary vector into another. For binary vectors $P = (p_1, p_2, ..., p_n)$ and $Q = (q_1, q_2, ..., q_n)$, it's defined as:

$$d_{\text{hamming}}(P, Q) = \sum_{i=1}^{n} \delta(p_i, q_i), \text{ where } \delta(p, q) = \begin{cases} 1 & \text{if } p \neq q \\ 0 & \text{if } p = q \end{cases}$$

- **Squared Distance**: This is simply the square of the Euclidean distance. It avoids computing the square root and can be computationally more efficient. For points $P = (A_p, B_p, ..., C_p)$ and $Q = (A_q, B_q, ..., C_q)$, it's defined as:

$$d_{\text{squared}}(P, Q) = \max\left(|A_P - A_Q|, |B_P - B_Q|, |C_P - C_Q|\right)$$

Let's consider a small dataset with three data points:

- $A = (1, 2)$
- $B = (3, 4)$
- $C = (5, 6)$

We'll calculate the distance between all pairs of these points using various distance metrics.

2.5.1 Hamming Distance

Hamming distance isn't applicable to this dataset because it's used for binary data.

2.5.2 Euclidean Distance

Euclidean distance between A and B:

$$d_{\text{euclidean}}(A, B) = \sqrt{(3-1)^2 + (4-2)^2} = 2\sqrt{2} \tag{2.1}$$

Euclidean distance between B and C:

$$d_{\text{euclidean}}(B, C) = \sqrt{(5-3)^2 + (6-4)^2} = 2\sqrt{2} \tag{2.2}$$

2.5.3 City Block Distance (Manhattan Distance)

City block distance between A and B:

$$d_{\text{city block}}(A, B) = |3 - 1| + |4 - 2| = 2 + 2 = 4 \tag{2.3}$$

City block distance between B and C:

$$d_{\text{city block}}(B, C) = |5 - 3| + |6 - 4| = 2 + 2 = 4 \tag{2.4}$$

2.5.4 Squared Euclidean Distance

Squared Euclidean distance between A and B:

$$d_{\text{squared}}(A, B) = (3 - 1)^2 + (4 - 2)^2 = 4 + 4 = 8 \tag{2.5}$$

Squared Euclidean distance between B and C:

$$d_{\text{squared}}(B, C) = (5 - 3)^2 + (6 - 4)^2 = 4 + 4 = 8 \tag{2.6}$$

2.6 Linear Classifier

A linear classifier is a method used in machine learning for separating data points into different classes. It does this by drawing a linear decision boundary, or a hyperplane, in the feature space.

The decision boundary of a linear classifier is defined by a discriminating function $f(\mathbf{X})$ that is a weighted sum of the features, subtracting a threshold θ:

$$f(\mathbf{X}) = \sum_i w_i x_i \tag{2.7}$$

where $\mathbf{X} = [x_1, x_2, ..., x_n]$ is the feature vector, $\mathbf{W} = [w_1, w_2, ..., w_n]$ are the weights, and n is the number of features.

For a given input $\mathbf{X}$, the classifier assigns it to one class if $f(\mathbf{X})$ is greater than zero, and another class otherwise:

- If $f(\mathbf{X}) > 0$, then $\mathbf{X}$ is classified as Class 1.
- If $f(\mathbf{X}) \leq 0$, then $\mathbf{X}$ is classified as Class 2.

Using matrix algebra, the discriminant function can be written more compactly as:

$$f(\mathbf{X}) = \sum_i w_i x_i - \theta \tag{2.8}$$

2.7 Controlling the Decision Boundary

- The decision boundary's position in the pattern space is controlled by two parameters: the slope of the line and the y-axis intercept.
- The slope of the decision boundary is determined by the weight vector.
- When the classifier output is zero ($f(\mathbf{X}) = 0$), this condition is known as the crossover point or boundary condition.
- Comparing the decision function to the equation of a straight line, the ratio of weight values w_1 and w_2 controls the slope, while the bias value, θ, controls the y-axis intercept.

2.8 Finding the Weight Vector

- Determining the appropriate weight vector is a critical but non-trivial task.
- The weight vector is typically found through iterative trial and error methods that modify weights based on an error function.
- This error function measures the discrepancy between the classifier's output and the desired response.

2.9 k-Nearest Neighbors (k-NN) Algorithm

The k-NN algorithm is a type of instance-based learning method used in machine learning. Given a new, unseen observation, it operates by finding the 'k' samples in the training dataset that are closest (in terms of a distance metric) to the new sample, and outputs the most common output variable among these 'k' nearest neighbors as the prediction.

The steps of the k-NN algorithm are as follows:

1. **Define the parameter k**: The number of nearest neighbors to consider.
2. **Calculate Distances**: For a new input, compute its distance from all samples in the training set. The distance could be calculated using different distance measures like Euclidean, Manhattan etc.

$$D(x, x_i) = \sqrt{(x_1 - x_{i1})^2 + (x_2 - x_{i2})^2 + \cdots + (x_n - x_{in})^2}$$

3. **Find Nearest Neighbors**: Identify 'k' samples in the training data that are closest to the new point.
4. **Vote for Labels**: Each of the 'k' nearest neighbors votes for their class label.
5. **Make a prediction**: Assign the new point the class that has the most votes.

Chapter 3

The Basic Neuron

3.1 Modeling the Single Neuron

A neuron in a neural network is a computational unit that takes a set of inputs, applies a transformation to them, and produces an output. The neuron can be modeled as follows:

- **Inputs**: Each neuron receives multiple inputs, represented as a feature vector $\mathbf{X} = [x_1, x_2, ..., x_n]$. Each input is associated with a weight $\mathbf{W} = [w_1, w_2, ..., w_n]$.
- **Weighted Sum**: The inputs and weights are multiplied element-wise and then summed to produce a single value, which is then offset by a bias term θ. The output of this operation is the net input, represented as:

$$z = \sum_i w_i x_i - \theta \tag{3.1}$$

- **Activation Function**: The net input is then passed through an activation function f, which determines the neuron's output. Commonly used activation functions include the sigmoid, hyperbolic tangent, and ReLU functions. The output is calculated as:

$$y = f(z) \tag{3.2}$$

3.1.1 Threshold Function

The Threshold function is a type of activation function used in artificial neurons and neural networks, that produces a binary output, typically either 0 or 1, based on whether the input exceeds a certain threshold value.

$$f(x) = \begin{cases} 0 & \text{if } x < \text{threshold} \\ 1 & \text{if } x \geq \text{threshold} \end{cases} \tag{3.3}$$

Alternative names for the Threshold function include:

- Step Function
- Heaviside Function

3.1.2 Difference between Activation Function and Threshold Function

The activation function and the threshold function both play a role in determining the output of a neuron in a neural network, but they are used in different contexts and have different characteristics.

- **Activation Function**: An activation function is a general term for a function that takes the weighted sum of the inputs of a neuron and transforms it into an output signal. It can be linear or non-linear. Examples include the Sigmoid function, Hyperbolic Tangent (tanh) function, and Rectified Linear Unit (ReLU) function.
- **Threshold Function**: The threshold function is a specific type of activation function that produces binary output (usually 0 or 1) depending on whether the input exceeds a certain threshold. The threshold function is also referred to as a step function or a binary step function.

In summary, while the threshold function can serve as an activation function, not all activation functions are threshold functions due to the variety of forms activation functions can take.

3.2 Perceptron Learning Algorithm

The perceptron is a simple supervised learning algorithm for binary classifiers. It is a type of linear classifier that makes its predictions based on a linear predictor function.

The steps of the Perceptron Learning Algorithm are as follows:

1. Initialize the weight vector and the bias with zero or small random numbers.
2. For each example in our training set, perform the following steps:
    - Compute the output value using the current weights and bias.
    - Update the weights and the bias if an error is made.

3. Repeat step 2 until the algorithm converges (no errors) or a predetermined number of epochs are completed.

The weight update in step 2 can be done using the following rule:

$$w_i = w_i + \Delta w_i \tag{3.4}$$

where:

$$\Delta w_i = \eta(y^{(j)} - \hat{y}^{(j)})x_i^{(j)} \tag{3.5}$$

Here, $y^{(j)}$ is the true class label, $\hat{y}^{(j)}$ is the predicted class label, $x_i^{(j)}$ is the i^{th} feature value, and η is the learning rate.

3.3 Adaptive Weights: Weight Updating Strategy

- **Standard Rule (Delta Rule):**

$$w(t+1) = w(t) + \eta(y - \hat{y})x$$

- **Widrow-Hoff Rule (Least Mean Square Rule):**

$$w(t+1) = w(t) + \eta(y - \hat{y})x(t)$$

- **Batch Update Rule:**

$$w(t+1) = w(t) + \eta \sum_{i \in D} (y_i - \hat{y}_i)x_i$$

Note: In these equations:

- $w(t)$ represents the weight vector at time t.
- η is the learning rate.
- y is the desired output.
- $\hat{y}$ is the actual output.
- x is the input vector.

- For the Batch Update Rule, the sum is over the entire dataset D.

This rule adjusts the weights in the direction of steepest descent of the error surface, aiming to find the minimum error.

3.4 Limitations of Perceptrons

While perceptrons are foundational to neural networks and have their uses, they also have certain limitations. These include:

1. **Binary Outputs**: Perceptrons can only output binary classifications (0 or 1), which limits their utility in multi-class classification problems.
2. **Linearly Separable Data**: The perceptron learning algorithm is only guaranteed to converge if the classes are linearly separable. If the data is not linearly separable, the algorithm will never converge.
3. **Single Layer**: A single-layer perceptron can only learn linearly separable patterns, as it essentially draws a single line (or hyperplane in higher dimensions) to separate the data. More complex, non-linear patterns require multi-layer perceptrons (i.e., neural networks with hidden layers).
4. **No Probability Estimates**: Perceptrons do not provide probability estimates for the predictions, which are often useful in many machine learning tasks.
5. **Inflexible Threshold**: In a perceptron, the threshold is fixed (i.e., it cannot be adjusted), which can limit the model's ability to learn complex data patterns.

3.5 Credit Assignment Problem in Perceptrons

The "credit assignment problem" in the context of perceptrons, or more generally in neural networks, refers to the challenge of determining which neurons or connections in the network are responsible for any given error in the output.

When an error occurs, it can be challenging to determine which parts of the network should be adjusted to reduce the error. Should the weights of a particular connection be modified? Should the activation function of a particular neuron be adjusted? Or is some combination of these changes needed? These are the questions that the credit assignment problem seeks to answer.

This problem is especially difficult in networks with hidden layers, since the relationship between the input, the weights, and the output is not directly observable. The backpropagation algorithm is a commonly used method to solve the credit assignment problem in multi-layer

perceptrons, as it effectively computes how much each neuron's weights contributed to the final error.

Chapter 4

The Multi-layer Perceptron

4.1 Backpropagation Algorithm for Multilayer Perceptron

Backpropagation is an efficient method used to calculate the gradient of the loss function in a neural network, which can then be used by a gradient descent algorithm to update the weights of the network. Here are the main steps of backpropagation for an MLP:

1. **Initialization**: Initialize the weights with small random numbers.
2. **Forward Pass**: For each input in the dataset:
   - Propagate the input forward through the network.
   - Calculate the output of each neuron from the input layer, through the hidden layers, to the output layer.

   The output of each neuron i in layer l can be calculated using:

$$y_i^l = f\left(\sum_j w_{ji}^l y_j^{l-1} + b_i^l\right) \tag{4.1}$$

   where f is the activation function, w_{ji}^l is the weight from neuron j in layer $l-1$ to neuron i in layer l, and b_i^l is the bias of neuron i in layer l.
3. **Backward Pass**: For each output neuron, calculate its error term and then propagate these errors back through the network to compute the error term for each hidden neuron. The error term δ_i^l of neuron i in layer l is calculated as follows:

- For the output layer ($l = L$):

$$\delta_i^L = (y_i^L - t_i) f'(h_i^L) \tag{4.2}$$

where t_i is the target output, and h_i^L is the weighted sum of inputs of the output neuron before activation.

- For the hidden layers ($l < L$):

$$\delta_i^l = \left(\sum_k w_{ik}^{l+1} \delta_k^{l+1} \right) f'(h_i^l) \tag{4.3}$$

4. **Weight Update**: Update each weight in the network using the error terms. The weights are updated according to the rule:

$$w_{ji}^l = w_{ji}^l - \eta \, y_j^{l-1} \delta_i^l \tag{4.4}$$

where η is the learning rate.

5. **Iteration**: Repeat the Forward Pass, Backward Pass, and Weight Update steps for the desired number of iterations or until the network performance meets the desired criteria.

4.2 Example of Forward and Backward Propagation

4.2.1 Dataset

We'll use a simple dataset with two training examples:

$$(x_1, y_1) = (0.5, 0.8), \quad (x_2, y_2) = (0.6, 0.9).$$

4.2.2 Initial Weights and Biases

Our initial weights and biases are:

$$W^{[1]} = \begin{bmatrix} 0.3 & 0.1 \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} 0.2 & 0.2 \end{bmatrix},$$

$$W^{[2]} = \begin{bmatrix} 0.4 & 0.5 \end{bmatrix}, \quad b^{[2]} = 0.3.$$

4.2.3 Forward Propagation

1. Calculate the weighted input to the hidden layer:

$$Z^{[1]} = W^{[1]}X + b^{[1]} = \begin{bmatrix} 0.3 \times 0.5 + 0.2 \\ 0.1 \times 0.5 + 0.2 \end{bmatrix} = \begin{bmatrix} 0.35 \\ 0.25 \end{bmatrix}.$$

2. Apply the activation function (sigmoid):

$$A^{[1]} = \frac{1}{1 + e^{-Z^{[1]}}} = \begin{bmatrix} \frac{1}{1+e^{-0.35}} \\ \frac{1}{1+e^{-0.25}} \end{bmatrix}.$$

3. Repeat for the output layer:

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}, \quad A^{[2]} = \frac{1}{1 + e^{-Z^{[2]}}}.$$

4.2.4 Backward Propagation

1. Compute the error at the output layer:

$$\delta^{[2]} = A^{[2]} - Y = A^{[2]} - \begin{bmatrix} 0.8 \\ 0.9 \end{bmatrix}.$$

2. Compute the error at the hidden layer:

$$\delta^{[1]} = (W^{[2]T}\delta^{[2]}) \odot A^{[1]} \odot (1 - A^{[1]}),$$

where "$\odot$" denotes element-wise multiplication.

3. Compute the derivatives and update the weights and biases:

$$W^{[l]} = W^{[l]} - \alpha\delta^{[l]}A^{[l-1]T}, \quad b^{[l]} = b^{[l]} - \alpha\delta^{[l]}.$$

4.3 Activation Functions

4.3.1 Rectified Linear Unit (ReLU)

Purpose: The ReLU function is used to introduce non-linearity into the network. It helps to mitigate the vanishing gradient problem, allowing the network to learn faster and perform better.

Equation:

$$f(x) = \max(0, x) \tag{4.5}$$

Advantages:

- Helps to alleviate the vanishing gradient problem.
- Computationally efficient.

Disadvantages:

- Neurons can "die" during training, i.e., they may stop outputting anything other than 0 if the weights are updated such that the weighted sum of the neuron's inputs is negative.

4.3.2 Sigmoid

Purpose: The sigmoid function is used to map the output of a neuron to a range between 0 and 1, making it useful for output neurons in networks intended for binary classification problems.

Equation:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4.6}$$

Advantages:

- Maps output to range (0, 1), which can be useful for interpreting the output as a probability.
- Smooth gradient, which can be beneficial in gradient descent.

Disadvantages:

- Suffers from the vanishing gradient problem, which can slow down learning or cause it to stop altogether.
- Outputs are not zero-centered.
- Computationally expensive due to the exponential operation.

4.3.3 Advantages of Sigmoid Function Over Hard Limiting Threshold Function

- Smooth, differentiable everywhere
- Maps inputs between 0 and 1

- Softer, gradual non-linearity
- Avoids issue of "dead neurons"

4.4 Equations in Multi-Layer Perceptron (MLP)

4.4.1 Input to Hidden Layer

In an MLP, the transformation from the input layer to the hidden layer is determined by a weight matrix, a bias vector, and an activation function. Given an input vector $\mathbf{x}$, the output $\mathbf{h}$ of the hidden layer can be calculated as:

$$\mathbf{h} = f(\mathbf{W}_{ih}\mathbf{x} + \mathbf{b}_h) \tag{4.7}$$

where $\mathbf{W}_{ih}$ is the weight matrix between the input and hidden layer, $\mathbf{b}_h$ is the bias vector of the hidden layer, and $f(\cdot)$ is the activation function.

4.4.2 Hidden to Output Layer

Similarly, the transformation from the hidden layer to the output layer is determined by another weight matrix, another bias vector, and typically another (or sometimes the same) activation function. Given the output $\mathbf{h}$ of the hidden layer, the output $\mathbf{y}$ of the MLP can be calculated as:

$$\mathbf{y} = g(\mathbf{W}_{ho}\mathbf{h} + \mathbf{b}_o) \tag{4.8}$$

where $\mathbf{W}_{ho}$ is the weight matrix between the hidden and output layer, $\mathbf{b}_o$ is the bias vector of the output layer, and $g(\cdot)$ is the activation function.

4.5 ADALINE and MADALINE

ADALINE (Adaptive Linear Neuron): It's an early single-layer artificial neural network. The model consists of multiple nodes (neurons) in a single layer, where each node takes in an input and a weight, multiplies them together, and passes the sum through a linear transfer function (identity function) to produce the output. The learning process in ADALINE uses the least mean squares (LMS) method, also known as the Widrow-Hoff learning rule, for updating the weights.

$$y = \sum_i w_i x_i$$

MADALINE (Multiple ADaptive LINear Elements): is an extension of ADALINE. It's essentially a three-layered network, created by adding an extra layer of ADALINE nodes on top of the original ADALINE model, hence making it a multilayer network. The learning algorithm used in MADALINE is the same as in ADALINE, but it uses a rule known as Rule II, which allows for the changing of adaptive weights and, if necessary, the polarity of units in the middle layer. MADALINE was one of the first neural networks to be applied in practical applications, particularly for pattern recognition tasks.

$$y_j = \text{sign}\left(\sum_i w_{ji} x_i\right) \quad \text{for } j = 1, 2, \ldots, m$$

4.6 Local Minima Problem and Solutions

4.6.1 Local Minima Problem

In the context of neural networks, the local minima problem refers to the training algorithm getting stuck in a local minimum of the loss function, instead of finding the global minimum. This can result in a model that is suboptimal.

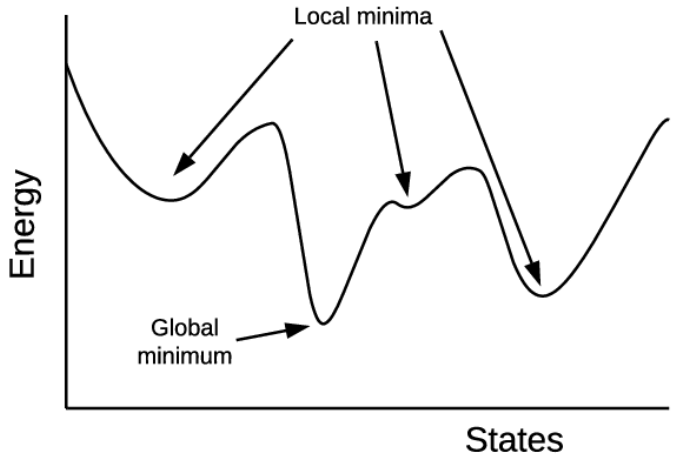


Figure 4.1: Illustration of Local vs Global Minima

4.6.2 Solutions to Local Minima Problem

1. **Random Initialization**: Initializing the weights to different values can result in the algorithm converging to different local minima.
2. **Momentum**: Adding a fraction of the update vector of the past time step to the current update vector can help the algorithm escape local minima.
3. **Learning Rate Schedule**: Adapting the learning rate over time can help the algorithm escape local minima.
4. **Using Advanced Optimizers**: Optimizers like Adam, RMSProp, etc. have mechanisms to avoid getting stuck in local minima.
5. **Adding Noise**: Injecting noise into either the input data or the gradient can help the model escape local minima.

4.7 Re-Learning in Multi-Layer Perceptrons (MLPs)

Re-learning is a process where an already trained MLP undergoes further training, typically to improve its performance based on new data, accommodate new classes, or adapt to changes in the data distribution.

The steps involved in re-learning are as follows:

1. **Retain the Learned Weights:** Keep the weights and biases from the previous learning phase.
2. **Introduce New Data:** Incorporate the new data into the training set. This could be new instances of existing classes, instances of new classes, or a combination of both.
3. **Training with New Data:** Use the same training algorithm (such as backpropagation) to train the network with the new data. During this process, the weights and biases are adjusted based on the errors the network makes when predicting the new data.

The main advantage of re-learning is that it allows the network to continuously adapt to changes or improvements in the data without having to undergo a complete retraining process.

4.8 Graceful Degradation in MLPs:

Graceful degradation in Multi-Layer Perceptrons (MLPs) refers to the network's ability to operate, albeit at reduced performance, even when some of its neurons or connections are not functioning.

This capability stems from:

1. **Distributed learning:** Information and learning are not concentrated in a single unit but distributed among many interconnected units.
2. **Redundancy:** Important features are often captured by multiple neurons and connections, offering a level of redundancy.
3. **Variety in training data:** Using diverse data helps the network generalize well, improving its resilience.

However, if too many units are damaged or lost, the performance degradation may become significant.

4.9 Fault Tolerance in Multi-Layer Perceptrons (MLPs)

Fault tolerance refers to the ability of an MLP to retain acceptable performance levels in the presence of faults or errors. These could be due to erroneous input data or even malfunctioning neurons within the network (e.g., neurons with incorrect weights or biases).

Fault tolerance in MLPs can be achieved through a variety of methods:

1. **Redundancy:** Additional neurons or layers are introduced to the network. These extras can step in if certain neurons or layers fail, allowing the network to maintain performance. The training process needs to ensure the redundancy is effective.
2. **Robust Training Techniques:** Certain training techniques, like noise injection during training, can enhance the robustness of the network, making it more resistant to faults.
3. **Self-Repair Mechanisms:** Some networks implement mechanisms that allow them to detect and repair faults, such as adjusting the weights and biases of malfunctioning neurons.

It is important to note that achieving high fault tolerance often requires a balance with other aspects of network performance and complexity. For instance, adding too much redundancy can make the network overly complex and could lead to problems such as overfitting.

4.10 Key Concepts in Model Training

Underfitting occurs when a model is too simple to capture the underlying structure of the data. The model has low variance but high bias. Mathematically, underfitting happens when the model error on the training set and the test set is high.

Overfitting happens when a model is excessively complex and starts to capture the noise in the data rather than the underlying structure. The model has low bias but high variance. Mathematically, overfitting occurs when the model error on the training set is low, but it's high on the test set.

Divergence in the context of training a neural network, refers to the situation where the error of the model on the training set keeps increasing during training instead of decreasing. This is often caused by an excessively high learning rate which results in the parameter updates overshooting the minimum of the loss function.

Question: what is the rule of select the node of output layer in backpropagration in MLP algorithm?
In a Multi-Layer Perceptron (MLP), the number of nodes in the output layer depends on the type of problem:

- **Regression Problems:** If the MLP is used for regression (predicting a continuous output), there is usually a single node.
- **Binary Classification Problems:** For binary classification (distinguishing between two classes), there is often one node. The output is the probability that the input example belongs to one class.
- **Multi-class Classification Problems:** For multi-class classification (distinguishing between more than two classes), there are typically as many nodes as there are classes. Each node outputs the probability that the input belongs to its corresponding class.

Chapter 5

Kohonen Self-Organising Network

5.1 Kohonen Self-Organizing Map Algorithm

The Kohonen Self-Organizing Map (SOM) is a type of neural network used for unsupervised learning(clustering). The algorithm is summarized as follows:

1. **Initialize** weights of neurons randomly.
2. **For** each training iteration:
   (a) **Present** a random input vector.
   (b) **Calculate** Euclidean distance from input vector to weight vector of each neuron.
   (c) **Identify** the Best Matching Unit (BMU), the neuron with the smallest distance to the input vector.
   (d) **Update** the weights of the BMU and its neighboring neurons to make them more similar to the input vector.
   (e) **Decrease** the learning rate and neighborhood radius over time.
3. **Repeat** the training iterations until convergence or the maximum number of iterations is reached.

This process maps the input vectors to a grid of neurons, preserving the topological properties of the input space in the grid.

5.2 Comparison between Self-Organizing Neural Networks and Supervised Neural Networks

SOM vs Supervised Neural Networks comparison is depicted on Table-5.1

Table 5.1: Comparison between Self-Organizing Neural Networks and Supervised Neural Networks

	Self-Organizing Networks	**Supervised Networks**
Learning Type	Unsupervised learning. No target output is provided.	Supervised learning. Target output is provided.
Training Data	Unlabeled data	Labeled data
Learning Process	The network learns the structure and patterns of the input data.	The network learns to map input data to given output.
Goal	Find hidden structure, clustering, or dimensionality reduction	Learn input-output mappings, classification or regression tasks
Example	Kohonen Self-Organizing Map (SOM)	Backpropagation, Support Vector Machine (SVM)
Adaptability	Adapts to find correlations and clusters in the data.	Adapts to minimize the difference between the actual and desired output.

5.3 Dimensionality Reduction

Dimensionality reduction refers to the process of converting data of very high dimensionality into data of much lower dimensionality such that each of the lower dimensions conveys much more information. This process is often necessary in applied machine learning, especially in situations where high-dimensional data causes problems (commonly referred to as the "curse of dimensionality"). Key points include:

Dimensionality Reduction:

- Simplifies high-dimensional data while retaining key information.
- Reduces computational cost and mitigates the risk of overfitting.
- Useful in visualizing high-dimensional data.

Dimensionality Reduction Using SOM:

Self-Organizing Maps (SOMs) are a type of artificial neural network designed for unsupervised learning. They work by creating a 'map' of the input data in a way that retains the topological (geometric and relational) properties of the data but in a lower-dimensional (usually 2D) form.

When an SOM is trained on input data, each input vector is matched to the most similar

neuron (referred to as the Best Matching Unit or BMU) in the SOM based on the weight vector of the neuron. This BMU along with its neighboring neurons then update their weights to be more similar to the input vector. This is done iteratively for all the data points in the dataset.

The 'map' that's formed as a result of this training process, is effectively a 2D representation of the high-dimensional input data. Different regions of the map correspond to different patterns or clusters in the input data.

- It achieves dimensionality reduction by preserving the topological properties of the input data.
- Project high-dimensional patterns onto a grid of neurons, each representing a region of the input space, thus effectively reducing the dimensionality.

5.4 Significance of Large Radius in Kohonen Networks

The radius in a Kohonen Self-Organizing Map (SOM) refers to the neighborhood around the Best Matching Unit (BMU), i.e., the neuron in the network that is most similar to the input vector. This radius determines which neurons in the network are updated during each iteration of the learning algorithm.

When the radius is large, more neurons are considered 'neighbors' of the BMU and are updated during each iteration. This means that the learning in the early stages of training is more global; the weights across a large portion of the network are adjusted, allowing the SOM to capture the broad structure of the input data.

However, as training proceeds, it is common to gradually decrease the radius. This focuses the learning on increasingly local regions around the BMU, allowing the network to fine-tune its representation of the data.

Therefore, a larger radius at the start of training can help ensure that the overall structure of the input data is captured. Gradually reducing the radius over time then allows the SOM to refine this structure and better learn the details of the data distribution.

5.5 Vector Quantization

Vector quantization is a process used in data compression where input vectors are grouped into clusters, each represented by a prototype vector (the weight vector of a neuron), effectively discretizing the input space.

In the context of Kohonen Self-Organizing Maps (SOMs), vector quantization refers to the process of assigning each input vector to the neuron (or "node") in the SOM whose weight

vector is most similar to the input vector. This neuron is known as the Best Matching Unit (BMU). The aim of this process is to map the high-dimensional input data onto the lower-dimensional grid of neurons in the SOM.

This process essentially compresses the information in the input data by representing similar input vectors with the same neuron in the SOM. The result is a quantized approximation of the input data that retains much of its structure while reducing its dimensionality and complexity.

5.6 Mexican Hat Function in Kohonen SOMs

The Mexican Hat function is used in Kohonen Self-Organizing Maps (SOMs) as a neighborhood function. Its purpose is to determine the amount of learning for neurons near the Best Matching Unit (BMU).

Key features of the Mexican Hat function:

1. **Excitatory and inhibitory effects:** The function's peak at the center symbolizes the highest weight update at the BMU. As we move away from the BMU, the weight updates decrease, modeling excitatory effects. However, at a certain distance, the function dips below zero, simulating inhibitory effects.

2. **Modeling lateral connections:** These excitatory and inhibitory effects effectively model the lateral connections between neurons in a Kohonen SOM.

It's important to note that weight initialization in a Kohonen SOM is usually random. The Mexican Hat function then governs how these weights are updated during learning.

5.7 Point Density Function in Self-Organizing Maps

In the context of Self-Organizing Maps (SOMs), the Point Density Function is a measure of how frequently each neuron (node) in the network is selected as the Best Matching Unit (BMU).

Each time a particular data point is input to the network during the training process, the BMU is the neuron whose weight vector is most similar to that data point. Thus, the Point Density Function essentially reflects the distribution of the input data as learned by the SOM.

By examining the Point Density Function of an SOM after training, we can get an idea of the density or frequency of the input data in the high-dimensional input space. Areas of the map with a high point density indicate regions of the input space where data points are concentrated, while areas with low point density indicate regions where data points are scarce.

Chapter 6

Hopfield Network

6.1 Hopfield Network

A Hopfield Network is a form of recurrent artificial neural network that serves as an auto-associative memory system.

6.1.1 Three Steps of Hopfield Network

1. Initialize the network: weights
2. Input unknown pattern
3. Iterate until convergence.

6.1.2 Pseudocode: Hopfield Network

Algorithm 2: Hopfield Network

Input: Unknown pattern V

Output: Recalled pattern

Initialize weights $W_{ij} = \sum_p V_{pi} \cdot V_{pj}$ for $i \neq j$, $W_{ii} = 0$

Apply unknown input pattern V_i

repeat

  foreach *neuron i in the network* **do**

    Compute net input $net_i = \sum_j W_{ij} \cdot V_j$

    Update neuron state $V_i = \text{sgn}(net_i)$, where $\text{sgn}(x)$ is the sign function

  end

until *network state does not change or maximum iterations reached*

6.2 Hopfield Network Training and Recall Process

Training Samples

Let's define four training samples, each one being a vector of length 5:

$$
\begin{aligned}
x_1 &= [1, -1, 1, -1, 1] \\
x_2 &= [-1, -1, -1, -1, -1] \\
x_3 &= [1, 1, -1, -1, -1] \\
x_4 &= [-1, 1, 1, 1, 1]
\end{aligned}
$$

6.2.1 Weight Matrix Calculation

We start by initializing the 5x5 weight matrix W to zeros. We then update the weights using the formula: $W = \sum_{i=1}^{4} x_i x_i^T - 4I$ where I is the 5x5 identity matrix.

After calculating, we get the weight matrix as follows (rounded to 2 decimal places):

$$
W = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & -2 & 2 & 2 \\
0 & -2 & 0 & 2 & 2 \\
0 & 2 & 2 & 0 & -2 \\
0 & 2 & 2 & -2 & 0
\end{bmatrix}
$$

6.2.2 Test Sample

Let's say our test sample is a slightly corrupted version of x_1:

$$x_{test} = [1, -1, 1, 1, 1]$$

6.2.3 Recall Process

For the recall process, we start with the corrupted test vector $x_{\text{test}} = [1, -1, 1, 1, 1]$. We then input this into the network and update it iteratively using the formula: $x_{\text{test}} = \text{sgn}(W \cdot x_{\text{test}})$, where sgn is the sign function that returns the sign of a number (1 for positive and -1 for negative values).

This process involves calculating the dot product of the weight matrix W and the test vector x_{test}, applying the sign function to the result, and then updating x_{test} with this new value.

We repeat this until x_{test} converges to one of the trained patterns. In our case, we expect it to converge to x_1.

$$\begin{aligned}
\text{Iteration 1:} \quad & W \cdot x_{\text{test}} = [0, 2, 2, -2, -2] \\
& \text{sgn}(W \cdot x_{\text{test}}) = [1, 1, 1, -1, -1] \\
& x_{\text{test}} = [1, 1, 1, -1, -1] \\
\text{Iteration 2:} \quad & W \cdot x_{\text{test}} = [0, 0, 0, 0, 0] \\
& \text{sgn}(W \cdot x_{\text{test}}) = [1, -1, 1, -1, 1] \\
& x_{\text{test}} = [1, -1, 1, -1, 1]
\end{aligned}$$

After two iterations, x_{test} becomes equal to x_1, demonstrating that the network has successfully learned and stored the patterns.

6.3 Features of the Hopfield Algorithm

- Fully connected, recurrent neural network.
- Trains with Hebbian learning rule.
- Stores binary or bipolar data.
- Robust against noisy inputs.
- Guarantees convergence to a local minimum.
- Simple implementation.

6.4 Nodes Required for Distinct Classes

A key characteristic of Hopfield networks is that the number of stably stored patterns is approximately 15% of the number of neurons in the network. Therefore, to store 150 distinct patterns (classes), we would need around 1000 nodes. This is calculated as follows:

$$\text{Required Nodes} \approx \frac{\text{Number of Classes}}{0.15} = \frac{150}{0.15} \approx 1000$$

6.5 Energy Landscape and Energy Function

Energy Landscape:
The energy landscape is a term used in the context of Hopfield networks and similar systems to describe the dynamic behavior of the system over time. The "Energy Landscape" can be seen as a map of all possible states of a Hopfield network. Each point on this map represents a different state of the network, and the 'height' of the point is determined by the energy of that state, as given by the energy function.

In this landscape, the network always tries to 'move' towards states of lower energy, similar to how a ball rolls downhill.

Energy Function:
In the context of Hopfield networks, the energy function is a mathematical function that assigns an 'energy' value to each state of the network. It is defined as:

$$E = -\frac{1}{2} \sum_{i} \sum_{j} w_{ij} V_i V_j$$

where w_{ij} are the weights, and V_i and V_j are the neuron states. In this system, states with lower energy are more 'stable' and are more likely to be 'remembered' by the network. When the network is given a starting state, it will update the states of the neurons in a way that 'lowers' the energy, until it reaches a state of minimum energy. The network uses this energy function to 'choose' which state to move to next - it will always try to move to a state of lower energy.

6.6 Storing Patterns in Hopfield Network

Storing patterns in a Hopfield network involves adjusting the weights of the neuron connections according to the patterns.

Given a pattern P, represented by a vector of neuron states, we store this pattern in the network by updating the weights using Hebb's Rule:

$$w_{ij} = \sum_{p} V_{pi} V_{pj}$$

Here, V_{pi} and V_{pj} represent the states of neurons i and j in pattern p, and the sum is over all patterns.

When a distorted pattern is presented to the network, the network evolves by updating the

states of the neurons as follows:

$$V_i = sgn(\sum_j w_{ij} V_j)$$

The network updates each neuron's state using this rule until it reaches a stable configuration, which corresponds to the recalled pattern.

6.7 symmetrically weighted input in Hopfield neural network

In a Hopfield network, the term *symmetrically weighted inputs* refers to the property where the weight of the connection from neuron i to neuron j (denoted as w_{ij}) is the same as the weight of the connection from neuron j to neuron i (denoted as w_{ji}). In other words, $w_{ij} = w_{ji}$. This symmetry is a key characteristic of Hopfield networks and is crucial for ensuring the stability of the network's dynamics.

6.8 Boltzmann Machine Learning Algorithm

A Boltzmann Machine is a type of stochastic recurrent neural network, and it's used for optimization problems. Below is the pseudocode for the Boltzmann Machine learning algorithm.

Algorithm 3: Boltzmann Machine Learning Algorithm

Result: Learned weights and thresholds

Initialization: $W_{ij} = 0$, $\theta_i = 0$;

while *not converged* **do**

1. Clamping phase: Fix the states of the visible units to match a data vector;
2. Perform Gibbs sampling until thermal equilibrium, compute $\langle v_i v_j \rangle_{data}$ and $\langle v_i \rangle_{data}$;
3. Free running phase: Allow all units to update, compute $\langle v_i v_j \rangle_{model}$ and $\langle v_i \rangle_{model}$;
4. Update weights and thresholds: $W_{ij} = W_{ij} + \eta(\langle v_i v_j \rangle_{data} - \langle v_i v_j \rangle_{model})$, $\theta_i = \theta_i + \eta(\langle v_i \rangle_{data} - \langle v_i \rangle_{model})$;

end

Chapter 7

Fuzzy Logic

7.1 Fuzzy Logic

Fuzzy logic is a reasoning system that mimics human decision making with approximate, not exact, solutions.

- Allows partial membership values
- Mimics human reasoning

Advantages:

- Tolerant of imprecise data
- Flexibility in decision-making
- Handles uncertainty effectively

Disadvantages:

- Not exact, only approximate
- Requires extensive computing hardware
- Difficult to construct rules
- Not ideal for simple tasks

7.1.1 Fuzzy Logic System Architecture

A Fuzzy Logic System typically consists of four main components:

1. **Fuzzification Interface:** Converts the crisp or precise input into fuzzy sets.

2. **Knowledge Base:** Consists of the database and the rule base.
3. **Inference Engine:** Applies fuzzy operators to the antecedents of the rules and then applies the implication method to the consequents.
4. **Defuzzification Interface:** Converted back into crisp values from fuzzy output.

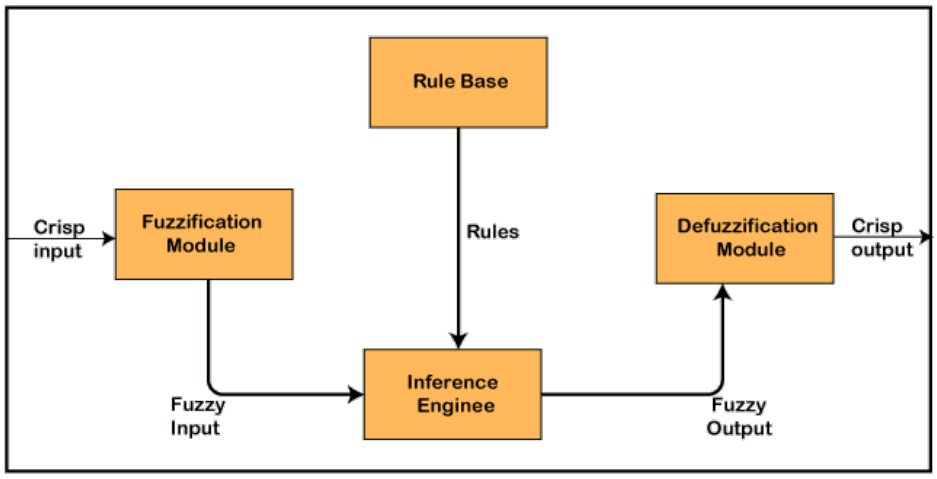


Figure 7.1: Illustration of Local vs Global Minima

Crisp Input/Output:

- Exact input/output
- Clear cut data
- No ambiguity
- Ex: temp 30 deg Celsius

Fuzzy set

- non exact value
- refer range
- Ex: temp is hot

7.2 Fuzzy Set Operations

- **Union:** Given two fuzzy sets A and B with membership functions $\mu_A(x)$ and $\mu_B(x)$ respectively, the union of A and B is a fuzzy set C with membership function $\mu_C(x) = \max\{\mu_A(x), \mu_B(x)\}$.

- **Intersection:** The intersection of A and B is a fuzzy set D with membership function $\mu_D(x) = \min\{\mu_A(x), \mu_B(x)\}$.
- **Complement:** The complement of a fuzzy set A is a fuzzy set E with membership function $\mu_E(x) = 1 - \mu_A(x)$.
- **Equality:** Two fuzzy sets A and B are considered equal if their membership functions are identical, i.e., $\mu_A(x) = \mu_B(x)$ for all x in the universe of discourse.

7.3 Defuzzification

Defuzzification is the process of converting a fuzzy output set into a crisp, single output that can be used in the real world.

7.3.1 Methods of Defuzzification

1. **Center of Gravity (or Centroid) Method:** The crisp output is the point where a vertical line would slice the aggregate set into two equal masses.
2. **Bisector of Area Method:** The crisp output is the point where a vertical line would slice the aggregate set into two equal areas.
3. **Mean of Maximum (MoM) Method:** The crisp output is the average of all maxima of the output fuzzy set.
4. **Maxima (or Maximum) Method:** The crisp output is the point of maximum membership of the aggregate set.
5. **Smallest of Maximum (SoM) Method:** The crisp output is the smallest value for which the output fuzzy set reaches its maximum.
6. **Largest of Maximum (LoM) Method:** The crisp output is the largest value for which the output fuzzy set reaches its maximum.

7.3.2 Defuzzification Methods Example

Consider a fuzzy set A with the domain $X = \{1, 2, 3, 4, 5\}$ and membership function values $\mu_A = \{0.2, 0.5, 1, 0.5, 0.2\}$.

The defuzzification methods can be applied as follows:

Center of Sums (COS):

$$COS = \frac{\sum_{x \in X} x \cdot \mu_A(x)}{\sum_{x \in X} \mu_A(x)} = \frac{1 \cdot 0.2 + 2 \cdot 0.5 + 3 \cdot 1 + 4 \cdot 0.5 + 5 \cdot 0.2}{0.2 + 0.5 + 1 + 0.5 + 0.2} = 3$$

Center of Gravity (COG) / Centroid of Area (COA):

$COG = COS = 3$ (for discrete data, COG = COS)

Weighted Average Method:

$$\text{Weighted Average} = \frac{\sum_{x \in X} x \cdot \mu_A(x)}{N} = \frac{1 \cdot 0.2 + 2 \cdot 0.5 + 3 \cdot 1 + 4 \cdot 0.5 + 5 \cdot 0.2}{5} = 2.08$$

7.4 Fuzzy Logic Controller (FLC)

A Fuzzy Logic Controller (FLC) is a system that uses fuzzy logic to make decisions and control complex processes.

7.4.1 FLC Design: Scenireo

Consider a Fuzzy Logic Controller (FLC) for an air conditioning system that considers the temperature and humidity to adjust the cooling.

Inputs

- Temperature: Can be "Cold", "Comfortable", or "Hot"
- Humidity: Can be "Low", "Normal", or "High"

Output

- Air Conditioner Setting: Can be "Low", "Medium", or "High"

Rules

1. IF temperature is "Hot" AND humidity is "High" THEN air conditioner setting is "High"
2. IF temperature is "Comfortable" AND humidity is "Normal" THEN air conditioner setting is "Medium"
3. IF temperature is "Cold" AND humidity is "Low" THEN air conditioner setting is "Low"

Sample Dataset

Temperature	Humidity
Hot	High
Comfortable	Normal
Cold	Low

Table 7.1: Fuzzy Logic design scenario

With the given rules and dataset, the FLC will set the air conditioner to "High" when it's hot and humid, "Medium" when the temperature and humidity are comfortable and normal respectively, and "Low" when it's cold and the humidity is low.

7.4.2 FLC Design: Process

Step 1: Define Fuzzy Sets and Membership Functions
Here-

- T: Temperature
- H: Humidity
- AC: AC Setting

Variable	**Set**	**Membership Function (Triangular)**
T	Low	$\mu_{T_{Low}}(T) = \max(\min(1, 1 - T/50), 0)$
T	Medium	$\mu_{T_{Medium}}(T) = \begin{cases} \max(\min(1, (T-50)/25), 0) & \text{for } T \geq 50 \\ \max(\min(1, (100-T)/25), 0) & \text{for } T < 50 \end{cases}$
T	High	$\mu_{T_{High}}(T) = \max(\min(1, (T-50)/50), 0)$
H	Similar definitions as T	
AC	Similar definitions as T	

Step 2: Fuzzify Inputs

Assume T=80 and H=60, we get:

$$\mu_{T_{Low}}(80) = 0,$$
$$\mu_{T_{Medium}}(80) = 0.6,$$
$$\mu_{T_{High}}(80) = 1,$$
$$\mu_{H_{Low}}(60) = 0,$$
$$\mu_{H_{Medium}}(60) = 0.8,$$
$$\mu_{H_{High}}(60) = 0.2$$

Step 3: Inference Rules

Applying the rules to compute minima:

$$\text{Rule 1 (IF T High AND H High THEN A High)} : \min(1, 0.2) = 0.2$$
$$\text{Rule 2 (IF T Medium AND H Medium THEN A Medium)} : \min(0.6, 0.8) = 0.6$$
$$\text{Rule 3 (IF T Low AND H Low THEN A Low)} : \min(0, 0) = 0$$

Step 4: Defuzzification

Assume we are using the centroid method for defuzzification. When we combine the output sets and determine the centroid, let's say it's 80 (for simplicity). So, the FLC suggests AC setting to 80 (a high setting).

Chapter 8

Genetic Algorithm

Genetic Algorithm tutorial course link: Genetic Algorithm Bangla Tutorial: Lecturelia

Genetic Algorithm note link: Ratul's Genetic Algorithm

8.1 Exercise: P8.1

In a three variable problem, the following variable bounds are specified:

$$-6 < x < 12,$$
$$0.002 \leq y \leq 0.004,$$
$$10^4 \leq z \leq 10^5.$$

What should be the minimum string length of any point (x,y,z) coded in binary string to achieve the following accuracy in the solution -

1. Two significant digits
2. Three significant digits

Solution:
We use the formula for string length L:

$$L = \lceil \log_2 \left(\frac{b-a}{e} \right) \rceil,$$

where a and b are lower and upper bounds, and e is the desired precision.

For **two significant digits** ($e = 0.01$), we calculate L for each variable:

- For x,

$$\begin{aligned} L_x &= \lceil \log_2 \left(\frac{12 - (-6)}{0.01} \right) \rceil \\ &= \lceil \log_2(1800) \rceil \\ &= 11. \end{aligned}$$

- For y,

$$\begin{aligned} L_y &= \lceil \log_2 \left(\frac{0.004 - 0.002}{0.01} \right) \rceil \\ &= \lceil \log_2(200) \rceil \\ &= 8. \end{aligned}$$

- For z,

$$\begin{aligned} L_z &= \lceil \log_2 \left(\frac{10^5 - 10^4}{0.01} \right) \rceil \\ &= \lceil \log_2(900000) \rceil \\ &= 20. \end{aligned}$$

For **three significant digits** ($e = 0.001$), we calculate L for each variable:

- For x,

$$\begin{aligned} L_x &= \lceil \log_2 \left(\frac{12 - (-6)}{0.001} \right) \rceil \\ &= \lceil \log_2(18000) \rceil \\ &= 15. \end{aligned}$$

- For y,

$$\begin{aligned} L_y &= \lceil \log_2 \left(\frac{0.004 - 0.002}{0.001} \right) \rceil \\ &= \lceil \log_2(2000) \rceil \\ &= 11. \end{aligned}$$

- For z,

$$\begin{aligned} L_z &= \lceil \log_2 \left(\frac{10^5 - 10^4}{0.001} \right) \rceil \\ &= \lceil \log_2(9000000) \rceil \\ &= 23. \end{aligned}$$

Note:

$$L = \lceil log_2 \left(\frac{b-a}{e} \right) \rceil \tag{8.1}$$

The formula for string length L is derived from binary encoding and precision requirements. To encode a real number range [a, b] into binary strings, we must decide on the number of bits to use. More bits allow higher precision (more representable values), while fewer bits lead to lower precision. If we wish to achieve a precision e, we need enough binary strings to cover the entire [a, b] range. Since the number of binary strings represented by L bits is 2^L, we need at least $\frac{b-a}{e}$ binary strings. By taking the base-2 logarithm, we get the required number of bits L. The ceiling function ensures L is an integer.

8.2 Exercise: P8.2

Repeat the adobe problem when ternary strings (with three alleles 0,1,2) are used instead of binary string.

Solution:

For a ternary string, the number of distinct strings represented by L digits is 3^L. We need at least $\frac{b-a}{e}$ ternary strings to cover the range [a, b] with precision e. Taking the base-3 logarithm of this quantity and rounding up to the nearest integer gives the required string length.

Thus, for two significant digits (i.e., $e = 0.01$ for x and z, and $e = 0.0001$ for y):

$$\begin{aligned} L_x &= \lceil \log_3 \left(\frac{12 - (-6)}{0.01} \right) \rceil = \lceil \log_3(1800) \rceil = 7 \\ L_y &= \lceil \log_3 \left(\frac{0.004 - 0.002}{0.0001} \right) \rceil = \lceil \log_3(20) \rceil = 3 \\ L_z &= \lceil \log_3 \left(\frac{10^5 - 10^4}{10^2} \right) \rceil = \lceil \log_3(900) \rceil = 7 \end{aligned}$$

So, the total length for two significant digits is $L_x + L_y + L_z = 7 + 3 + 7 = 17$.

For three significant digits (i.e., $e = 0.001$ for x and z, and $e = 0.00001$ for y):

$$L'_x = \lceil \log_3 \left(\frac{12 - (-6)}{0.001} \right) \rceil = \lceil \log_3(18000) \rceil = 9$$
$$L'_y = \lceil \log_3 \left(\frac{0.004 - 0.002}{0.00001} \right) \rceil = \lceil \log_3(200) \rceil = 5$$
$$L'_z = \lceil \log_3 \left(\frac{10^5 - 10^4}{10} \right) \rceil = \lceil \log_3(9000) \rceil = 9$$

So, the total length for three significant digits is $L'_x + L'_y + L'_z = 9 + 5 + 9 = 23$.

8.3 Exercise: P8.3

We want to use Genetic Algorithm (GA) to solve the following nonlinear programming problem:

$$\begin{aligned}
&\text{minimize } (x_1 - 2.5)^2 + (x_2 - 5)^2 \\
&\text{subject to:} \\
&5.5x_1 + 2x_2^2 - 18 \leq 0 \\
&0 \leq x_1, x_2 \leq 5
\end{aligned}$$

We decide to give three and two decimal places of accuracy to variables x_1, x_2 respectively.

1. How many bits are required for coding the variables?
2. Write down the fitness function which you would be using in reproduction.

Solution:
For the given constraints, we have the following limits for x_1 and x_2:

For $x_1 = 0$:

$$2x_2^2 \leq 18$$

Hence,

$$x_2 \leq \sqrt{9} = 3$$

For $x_1 = 5$:

$$5.5 \times 5 + 2x_2^2 \leq 18$$

Solving this we get

$$x_2^2 \leq -\frac{55}{4}$$

This does not have a real solution, hence x_1 cannot take the value of 5.

Similarly, for $x_2 = 0$ and $x_2 = 5$, we get the corresponding limits for x_1. Thus, we have the limits of x_1 and x_2 as:

$$0 \leq x_1 \leq \frac{18}{5.5} \quad \text{and} \quad 0 \leq x_2 \leq 3$$

1. Number of bits for coding variables:

The number of bits required, L, for a variable can be calculated as:

$$L = \lceil \log_2 \left(\frac{b-a}{e} \right) \rceil$$

where a and b are the lower and upper bounds of the variable, and e is the required precision.

For x_1 ($a = 0$, $b = \frac{18}{5.5}$, $e = 0.001$):

$$L_{x_1} = \lceil \log_2 \left(\frac{\frac{18}{5.5} - 0}{0.001} \right) \rceil = 12$$

For x_2 ($a = 0$, $b = 3$, $e = 0.01$):

$$L_{x_2} = \lceil \log_2 \left(\frac{3-0}{0.01} \right) \rceil = 9$$

So, a total of $L_{x_1} + L_{x_2} = 12 + 9 = 21$ bits are required to code the variables with the specified precision.

2. Fitness Function:

The given optimization problem is a minimization problem. Therefore, the fitness function could be the reciprocal of the objective function. However, to avoid division by zero, we often add 1 to the denominator. Hence, the fitness function could be written as:

$$f(x_1, x_2) = \frac{1}{1 + (x_1 - 2.5)^2 + (x_2 - 5)^2}$$

This fitness function ensures that smaller values of the objective function result in higher fitness.

8.4 Exercise: P8.4

Consider the following population of binary strings for a maximization problem.

String	**Fitness**
01101	5
11000	2
10110	1
00111	10
10101	3
00010	100

Table 8.1: Population of Binary Strings

Find out the expected number of copies of the best string in the above population of the mating pool under:

1. Roulette wheel selection.
2. Tournament selection.

If only the reproduction operator is used, how many generations are required before the best individual occupies the complete population under each selection operator?

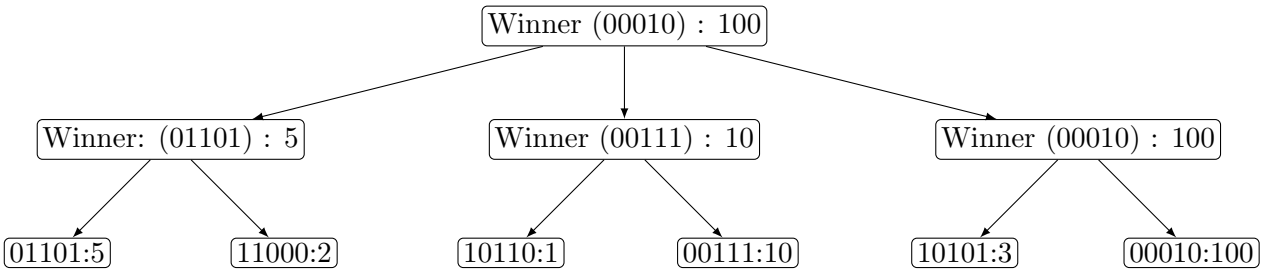
Solution:

1. Roulette wheel selection:

String	Fitness	Percentage	Probability	Expected Count	Actual Count
01101	5	4.13%	0.0413	0.2478	0
11000	2	1.65%	0.0165	0.099	0
10110	1	0.826%	0.00826	0.04956	0
00111	10	8.26%	0.0826	0.4956	1
10101	3	2.48%	0.0248	0.1488	0
00010	100	82.64%	0.8264	4.958	5

In Roulette wheel selection, the expected number of copies of a string in the next generation is proportional to its fitness. The total fitness of the population is $5+2+1+10+3+100 = 121$. The best string has a fitness of 100. So, the expected number of copies is $\frac{100}{121} \times 6 \approx 4.958$.

2. Tournament selection:
In tournament selection, the fittest individual always wins, so the best string will be selected in every tournament. Therefore, the expected number of copies of the best string is equal to the population size, which is 6 in this case.



As for the number of generations required before the best individual occupies the complete population, in the case of roulette wheel selection, this would be random. However, it's likely to occur quicker in the tournament selection since the fittest individual is always selected.

8.5 Exercise: P9.1

We want to use Genetic Algorithm (GA) to solve the following nonlinear programming problem:

$$\text{minimize } (x_1 - 1.5)^2 + (x_2 - 4)^2$$

subject to:

$$4.5x_1 + x_2^2 - 18 \leq 0$$

$$2x_1 - x_2 - 1 \geq 0$$

$$0 \leq x_1, x_2 \leq 4$$

Show calculations for three generations. Use crossover probability as 80% and a mutation probability of 3% ?

Solution:

1. Constraints for the decision variables:

First, let's solve the inequalities to find the actual range for x_1 and x_2.

The first inequality is:

$$4.5x_1 + x_2^2 - 18 \leq 0$$

Let's find the maximum possible value for x_1 by assuming $x_2 = 0$ (since this will give the minimum value for the x_2^2 term):

$$4.5x_1 - 18 \leq 0 \Rightarrow x_1 \leq \frac{18}{4.5} = 4$$

Similarly, we can find the minimum possible value for x_1 by assuming $x_2 = 4$ (since this will

give the maximum value for the x_2^2 term):

$$4.5x_1 + 4^2 - 18 \geq 0 \Rightarrow x_1 \geq \frac{18 - 4^2}{4.5} = 0.5$$

So, the actual range for x_1 from this inequality is $0.5 \leq x_1 \leq 4$.

Now let's look at the second inequality:

$$2x_1 - x_2 - 1 \geq 0$$

We find the minimum possible value for x_2 by assuming $x_1 = 0$ (since this will give the minimum value for the $2x_1$ term):

$$-x_2 - 1 \geq 0 \Rightarrow x_2 \leq -1$$

But since x_2 cannot be negative according to the problem's constraints, x_2 must be 0. So, the actual range for x_2 from this inequality is $0 \leq x_2 \leq 4$.

So, after considering both inequalities and the constraints in the problem, the actual ranges are:

$$0.5 \leq x_1 \leq 4$$

$$0 \leq x_2 \leq 4$$

2. Number of bits for coding variables:

Assuming the precision of 2 decimal places for x_1 and x_2, the number of bits required can be calculated using the formula:

$$L = \lceil \log_2 \left(\frac{b - a}{e} \right) \rceil$$

where a and b are the lower and upper bounds of the variable, and e is the required precision.

For x_1 ($a = 0.5$, $b = 4$, $e = 0.01$):

$$L_{x_1} = \lceil \log_2 \left(\frac{4 - 0.5}{0.01} \right) \rceil = 9$$

For x_2 ($a = 0$, $b = 4$, $e = 0.01$):

$$L_{x_2} = \lceil \log_2 \left(\frac{4 - 0}{0.01} \right) \rceil = 8$$

Therefore, We'll assume that x_1 and x_2 are discretized to the nearest 0.01, which will require us to represent them as integers from 0 to 400. For simplicity, we'll assume we're using a

fixed-length binary encoding for x_1 and x_2 of 9 bits each.

Initial Population We'll start with a hypothetical initial population of four individuals:

$$
\begin{aligned}
p_1 &: 001011101\ 110011011 \\
p_2 &: 010110010\ 101101110 \\
p_3 &: 110100101\ 011001100 \\
p_4 &: 101001110\ 100101011
\end{aligned}
$$

Fitness Calculation We'll calculate the fitness of these individuals using the objective function:

$$f(x_1, x_2) = \frac{1}{1 + (x_1 - 1.5)^2 + (x_2 - 4)^2}$$

For simplicity, we'll only show the calculation for p_1:

$$
\begin{aligned}
x_{1,p_1} &= \text{int}(001011101)_2 = 93 \times 0.01 = 0.93 \\
x_{2,p_1} &= \text{int}(110011011)_2 = 403 \times 0.01 = 4.03 \\
f(x_{1,p_1}, x_{2,p_1}) &= \frac{1}{1 + (0.93 - 1.5)^2 + (4.03 - 4)^2} = 0.47
\end{aligned}
$$

Selection For selection, we'll assume a simple roulette wheel selection based on the calculated fitness.

Crossover With a crossover probability of 80%, we'll perform crossover for some pairs of selected individuals. The point of crossover is chosen randomly.

Mutation With a mutation probability of 3%, we'll perform mutation on some of the offsprings obtained from crossover. Mutation is done by randomly flipping some bits.

New Generation The new generation is formed by replacing some of the least fit individuals with the offsprings obtained from crossover and mutation.

This process is then repeated for the next generations.