Chapter 2

Computational Intelligence

"If computers get too powerful, we can organize [sic] them into a committee – that will do them in." — Bradley's Bromide

The formulation of a precise definition for Computational Intelligence (CI) and how it relates to the broader Artificial Intelligence (AI) field is a challenging task. Arguably, CI comprises of those paradigms in AI that relate to some kind of biological or naturally occurring system. General consensus suggests that these paradigms are neural networks, evolutionary computing, swarm intelligence and fuzzy systems [29, 31, 88, 130]. Neural networks are based on their biological counterparts in the human nervous system. Similarly, evolutionary computing draws heavily on the principles of Darwinian evolution observed in nature. Swarm intelligence, in turn, is modelled on the social behaviour of insects and the choreography of birds flocking. Finally, human reasoning using imprecise, or fuzzy, linguistic terms is approximated by fuzzy systems.

Figure 2.1 shows these four primary branches of CI and illustrates that hybrids between the various paradigms are possible. Another, more precise, definition describes CI as the study of adaptive mechanisms to enable or facilitate intelligent behaviour in complex and changing environments [31]. Yet there are other AI approaches, that satisfy both this definition as well as the requirement of modelling some naturally occurring phenomenon, that do not fall neatly into one of the paradigms mentioned thus far. Could it be argued that the definition for CI is in itself complex, changing and fuzzy? A more pragmatic approach might be to specify the classes of problems that are of interest without being too concerned about whether or not the solutions to these problems satisfy any constraints implied by a particular definition for CI.

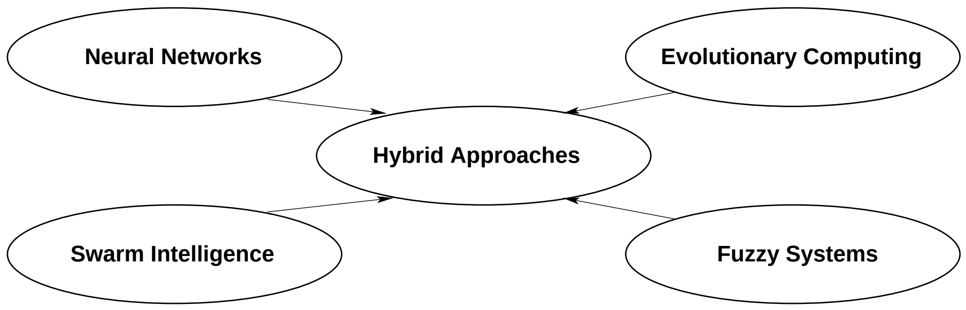

Figure 2.1: Computational Intelligence Paradigms

The following section identifies and describes four primary problem classes for CI techniques. A compendious overview of the main concepts behind each of the widely recognised CI paradigms is presented in Sections 2.2 through 2.5. Further, paradigms that are not generally recognised as CI, but that arguably also classify as such are mentioned in Section 2.6. Examples of hybrid approaches are given in Section 2.7. Finally, a discussion, in Section 2.8, concludes with some software implementation requirements made apparent by the contents of this chapter.

2.1 Problem Classes

Optimisation, defined in Section 2.1.1, is undoubtedly the most important class of problem in CI research, since virtually any other class of problem can be re-framed as an optimisation problem. This transformation, particularly in a software context, may lead to a loss of information inherent to the intrinsic form of the problem. The discussion in Section 2.8 illustrates how these intrinsic features can be exploited in software.

Section 2.1.2 discusses the well known travelling salesman problem as a model representative for the NP-Complete class of problems that are generally thought to be intractable. Function learning and classification, which are characteristic of supervised learning, are presented in Section 2.1.3. Finally, unsupervised learning is represented by clustering in Section 2.1.4.

2.1.1 Optimisation

The process of seeking out values for variables that either minimise or maximise some objective function is known as optimisation [12]. Stated formally, for the case of minimisation:

$$\textit{Given}: \quad f : \mathbb{S} \rightarrow \mathbb{R},\ \text{find } \mathbf{x}^* \in \mathbb{S} \text{ for which } f(\mathbf{x}^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{S} \tag{2.1}$$

where $\mathbb{S}$ represents the search domain which is typically, but but not necessarily, $\mathbb{R}^n$. The minimiser, $\mathbf{x}^*$, is the solution to the minimisation problem defined by the objective function f. The dual problem does not require separate discussion, since, in general, finding the maximiser for an objective function $g : \mathbb{S} \rightarrow \mathbb{R}$ is exactly the same as finding the minimiser for $f : \mathbb{S} \rightarrow \mathbb{R}$ with $f(\mathbf{x}) = -g(\mathbf{x})$.

When the objective function is defined for a search domain of $\mathbb{R}^n$, further equality and inequality constraints may be defined to restrict the feasible region in which solutions are considered. The constrained optimisation problem is defined formally as follows:

$$\textit{Given}: \quad f : \mathbb{R}^n \rightarrow \mathbb{R},\ \text{find } \mathbf{x}^* \in \mathbb{R}^n \text{ for which } f(\mathbf{x}^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{R}^n \tag{2.2}$$

$$\text{subject to} \quad p_i(\mathbf{x}) = 0,\ i \in \{\mathbb{Z} \mid 1 \leq i \leq r\} \tag{2.3}$$

$$q_j(\mathbf{x}) \geq 0,\ j \in \{\mathbb{Z} \mid 1 \leq j \leq s\} \tag{2.4}$$

where $p_i(\mathbf{x})$ and $q_j(\mathbf{x})$ are respectively, r equality and s inequality constraint functions on the components of the vector $\mathbf{x} \in \mathbb{R}^n$. Constraints of the form $a \leq x_k \leq b$ for $k \in \{\mathbb{Z} \mid 1 \leq k \leq n\}$ can be rewritten as two instances of the single sided inequality constraint of Equation (2.4), namely $q_a(\mathbf{x}) = x_k - a$ and $q_b(\mathbf{x}) = -x_k + b$.

Many algorithms for performing optimisation are designed to be applied to unconstrained optimisation problems, so it is desirable to be able to convert a constrained problem into the form of Equation (2.1) with $\mathbb{S} = \mathbb{R}^n$. A simple method to achieve this is to add to the objective function a suitable penalty term encapsulating the constraints. Thus, the function under optimisation becomes $f(\mathbf{x}) = g(\mathbf{x}) + P(\mathbf{x})$ where $P(\mathbf{x})$ is the penalty term.

Another technique, known as Lagrange's method [69], can be used to convert a constrained problem with equality constraints of the form in Equation (2.3) to an unconstrained problem. The Lagrange function is defined as:

$$\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) - \sum_{i=1}^{r} \lambda_i p_i(\mathbf{x}) \tag{2.5}$$

where $f(\mathbf{x})$ and $p_i(\mathbf{x})$ are the same as in Equation (2.2) and (2.3) respectively and the λ_i constants are known as Lagrange multipliers. At the optimal point of intersection the constraint and the objective functions are tangent to each other and so $\nabla f(\mathbf{x}) = \lambda_i \nabla p_i(\mathbf{x})$ provided that $\nabla p_i(\mathbf{x}) \neq 0$. Given Equation (2.5), this is true if and only if $\nabla \mathcal{L}(\mathbf{x}, \lambda) = 0$ so solving the following yields a solution to the original constrained problem:

$$\frac{\delta \mathcal{L}}{\delta x_k} = \frac{\delta \mathcal{L}}{\delta \lambda_i} = 0,\ i \in \{\mathbb{Z} \mid 1 \leq i \leq r\},\ k \in \{\mathbb{Z} \mid 1 \leq k \leq n\} \tag{2.6}$$

which defines a system of $r + n$ equations that can be cast into an unconstrained optimisation problem by minimising the SSE (Sum Squared Error) defined by:

$$f(\mathbf{x}, \lambda) = \sum_{k=1}^{n} \left(\frac{\delta \mathcal{L}}{\delta x_k}\right)^2 + \sum_{i=1}^{r} \left(\frac{\delta \mathcal{L}}{\delta \lambda_i}\right)^2 \tag{2.7}$$

where the point $(\mathbf{x}, \lambda)$ can be considered as a single vector argument to a function of the form $f(\mathbf{x})$ in Equation (2.1) with $\mathbb{S} = \mathbb{R}^{r+n}$. Inequality constraints can be handled in a similar fashion by introducing slack variables into a modified Lagrangian:

$$\mathcal{L}(\mathbf{x}, \lambda, \mu) = f(\mathbf{x}) - \sum_{i=1}^{r} \lambda_i p_i(\mathbf{x}) - \sum_{j=1}^{s} \mu_j (q_j(\mathbf{x}) - e_j) \tag{2.8}$$

where $q_j(\mathbf{x})$ is the same as in Equation (2.4), the μ_j constants are additional Lagrange multipliers and e_j is the slack variable corresponding to the j^{th} inequality constraint.

Optimisation can be further extended into the multi-objective case where the task is to satisfy multiple, possibly conflicting, objectives simultaneously [73]. For example, it may be required that cost be minimised while at the same time benefit is maximised. Some kind of trade off is required when objectives such as these clash, since optimising one necessarily causes deterioration of another. Generally, the goal is to find representative points belonging to the, possibly infinite, pareto optimum set of minimisers given a set of objective functions. A pareto [38], or non-dominated, point is a minimiser for which none of the objectives can be further improved without adversely affecting another. Each of these pareto minimisers represents a different trade off between objectives.

Multi-objective minimisation is formally stated as:

$$\begin{aligned}
\textit{Given}: &\quad F(\mathbf{x}) = \{f_k(\mathbf{x}) \mid f_k : \mathbb{S} \to \mathbb{R}\},\ k \in \{\mathbb{Z} \mid 1 \leq k \leq m\} \\
\text{find} &\quad X^* = \{\mathbf{x}^* \in \mathbb{S} \mid F(\mathbf{x}^*) \preccurlyeq F(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{S}\} \\
\text{where} &\quad F(\mathbf{x}) \preccurlyeq F(\mathbf{y}) \iff (\forall_i)(f_i(\mathbf{x}) \leq f_i(\mathbf{y})) \wedge (\exists_i)(f_i(\mathbf{x}) < f_i(\mathbf{y}))
\end{aligned} \tag{2.9}$$

where X^* is a representative set of non-dominated minimisers and $F(\mathbf{x})$ is the set of m objective functions. The expression $F(\mathbf{x}^*) \preccurlyeq F(\mathbf{x})$ denotes that $\mathbf{x}^*$, a pareto minimiser, dominates the point $\mathbf{x}$ which is not an element of the pareto set. Once again, the search domain $\mathbb{S}$ may be $\mathbb{R}^n$ and further constrained by Equations (2.3) and (2.4).

If only a single solution in the pareto set is required then multi-objective optimisation can be converted into a single objective optimisation problem of the form in Equation (2.1) by defining the objective as:

$$f(\mathbf{x}) = \sum_{k=1}^{m} w_k f_k(\mathbf{x}) \tag{2.10}$$

which is simply a weighted sum over the set of objective functions that comprise $F(\mathbf{x})$. By varying the weights w_k and performing sequential optimisation passes multiple solutions in the pareto set may be obtained.

2.1.2 NP-Complete Problems

The Travelling Salesman Problem (TSP) [52], a well known problem in computer science, belongs to the NP-Complete class of problems and has been chosen for discussion as a representative for its class. The best known deterministic algorithms able to solve problems of this class execute in exponential-time, or worse, in proportion to the amount of input data.

However, they all have Non-deterministic Polynomial-time (NP) solutions that, in order to yield correct results, require guessing correctly at every decision point during execution by means of some magical non-deterministic process. While such a magical algorithm does not have much practical use, this property does at least guarantee the existence of a short certificate that can be used to validate whether a given solution is correct or not. No polynomial-time deterministic algorithms are known to exist for these problems and as such they are considered to be intractable.

Furthermore, a subset of these problems known as NP-Complete are all polynomial-time reducible amongst themselves, meaning that finding an effective solution to one problem in NP-Complete implies having an effective solution to all those in NP-Complete. Certain CI algorithms, which are by their nature non-deterministic, can be applied in an attempt to yield approximate solutions, given large data sets, in a reasonable amount of time.

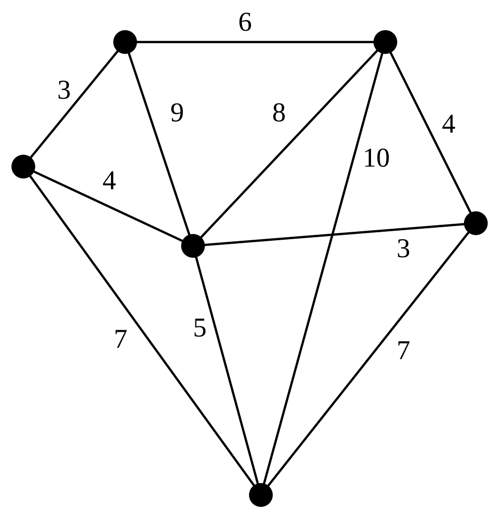

Figure 2.2: Example TSP Network (not to scale)

Problems in NP-Complete include knapsack packing, scheduling, graph colouring and testing the satisfiability of propositional calculus formulae amongst many other distinct problems. Some of these appear to be toy problems, such as the monkey puzzle problem [52], while others have important real world applicability. However, due to their polynomial-time inter-reducibility, all of them are actually of relatively equivalent importance.

In particular, the TSP has real world application in route optimisation, circuit design and the programming of industrial robots [52]. Moreover, the TSP is an ideal candidate for discussion, because it admits an interesting ant system solution (refer to Section 2.4.2) and, as described shortly, can also be cast into a constrained optimisation problem, as defined in the previous section.

The TSP concerns a salesman that must travel from city to city selling his wares before returning back to his city of origin. Each city must be visited exactly once and the distance travelled must be minimised. The problem can be characterised by a graph where each vertex represents a city while the edges correspond to the possible routes between cities and their associated costs. The goal is to determine the shortest closed tour that passes through each of the nodes in the graph for a given network. Figure 2.2 shows a possible network of cities while Figure 2.3 illustrates the optimal tour for that network which is of length 28.

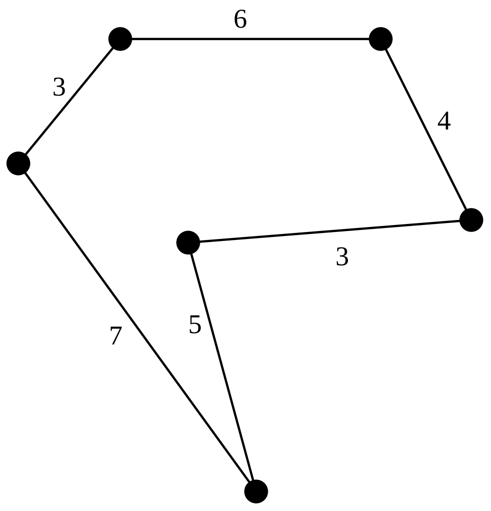

Figure 2.3: TSP Optimal Tour (length = 28)

By imposing an arbitrary ordering from 1 to n on the cities the problem can be redefined as determining the permutation π of visits that yield a minimal length tour. The problem is then reduced to the following constrained optimisation problem [83]:

$$\textit{Given}: \quad f(\mathbf{x}) = \sum_{i,j} c_{i,j} x_{i,j}, \; i,j \in \{\mathbb{Z} \mid 1 \le i,j \le n\} \tag{2.11}$$

$$\text{find} \quad \mathbf{x}^* \in \mathbb{Z}^{n\times n} \text{ for which } f(\mathbf{x}^*) \le f(\mathbf{x}), \; \forall \mathbf{x} \in \mathbb{Z}^{n\times n}$$

$$\text{subject to} \quad \sum_{k=1}^{n} x_{k,i} - 1 = 0 \text{ and } \sum_{k=1}^{n} x_{i,k} - 1 = 0 \tag{2.12}$$

$$x_{i,j} - 1 \le 0 \text{ and } -x_{i,j} \le 0 \tag{2.13}$$

$$u_i - u_j + n x_{i,j} - n + 1 \le 0 \text{ for } j \ne 1 \tag{2.14}$$

where $c_{i,j}$ is the cost of travelling from city i to j. In general, $c_{i,j} = c_{j,i}$ is not necessarily true, $c_{i,j} = \infty$ if no route from i to j exists, and $c_{i,j} = 0$ whenever $i = j$. Equation (2.13) restricts the $x_{i,j}$ to the boolean values 0 and 1 so that $x_{i,j} = 1$ can be taken to mean that city j is visited immediately after i and Equation (2.12) expresses that exactly one city just before and exactly one city just after the i^{th} city is visited. By defining $\pi(u_i) = i$, so that $u_i = j$ implies that i is the j^{th} city visited, a single closed tour is guaranteed by Equation (2.14). Together these constraints ensure that $x_{i,j} = 1 \iff \pi(i) = j$ and $x_{i,j} = 0 \iff \pi(i) \ne j$ when Equation (2.11) is minimised.

2.1.3 Supervised Learning

Supervised learning is the process of determining the intrinsic characteristics of a system using only examples of its operation [84]. The most generic form of supervised learning is function approximation, stated formally:

$$\begin{aligned} \textit{Given}: \quad & P = \{(\mathbf{x},\ \mathbf{t}) \mid \mathbf{x} \in \mathbb{S},\ \mathbf{t} \in \mathbb{T}\} \\ \text{find} \quad & f : \mathbb{S} \to \mathbb{T} \text{ such that } f(\mathbf{x}) \approx \mathbf{t},\ \forall (\mathbf{x},\ \mathbf{t}) \in P \end{aligned} \tag{2.15}$$

where P is a set of example patterns that demonstrate the operation of the system described by the function f. The pair $(\mathbf{x},\ \mathbf{t})$ is known as a training pattern where $\mathbf{x}$ is an input to the system under learning and $\mathbf{t}$ is the target output. $\mathbb{S}$ and $\mathbb{T}$ may be any domains. The process is called supervised learning because target values are provided for given inputs by some external "teacher" that understands the working of the system.

Care must be taken to ensure that the learning process does not over-fit the data [42]. Over-fitting may occur when the target function is afforded more degrees of freedom or less example patterns than are necessary to describe the system under learning. Under these circumstances the function may fit noise inherent in the data set or other very specific features that have no causal relation to the intrinsic characteristics of the system. Conversely, under-fitting occurs when the target function is not afforded enough degrees of freedom to properly model the underlying data.

The goal is to find a function that has good generalisation ability. This is measured by the ability of the learned function to correctly approximate the target output for inputs that the learning process was not exposed to. For this reason, the example patterns are typically partitioned into separate training and validation sets. Learning is performed using the training set while the validation set is used to test for over-fitting and generalisation ability. An over-fitted function will correctly model the training set while performing poorly on the validation set. On the other hand, a function with the ability to generalise well properly describes the intrinsic characteristics of the system under learning.

Supervised learning manifests itself in many forms including classification, pattern recognition and control problems. For classification problems, the function f in Equation (2.15) is a labelling function that assigns a class to an input pattern where $\mathbb{T}$ is some set of classes. Pattern recognition is just a special case of classification problem.

For example, in handwriting recognition, input patterns might correspond to bitmaps of hand written characters and the set of classes consists of alphanumeric assignments to those bitmaps. In control problems the function relates the sensory input of a system under control to the required output actions.

By defining a suitable parameterisation τ that describes the composition of the function f in Equation (2.15), supervised learning can be reduced to a minimisation problem of the form in Equation (2.1) as follows:

$$g(\tau) = \sum_{i=1}^{n} (\mathbf{t}_i - f(\mathbf{x}_i))^2, \text{ where } \tau \Rightarrow f \tag{2.16}$$

so that $g(\tau)$ is the SSE over the n training patterns, with $t \in \mathbb{R}$, for a function $f : \mathbb{S} \to \mathbb{R}$ implied by the parameterisation τ. Any suitable distance based metric can be used to support targets having arbitrary domains.

There are many ways to define the parameterisation τ. Supervised learning neural networks define very specific functions that are parameterised by weights (refer to Section 2.2). As another example, under the assumption that $\mathbf{x} \in \mathbb{R}^m$ and that the function can be approximated by a polynomial of degree n in each dimension, the following is a suitable definition:

$$f(\mathbf{x}, \tau) = \sum_{i=1}^{m} \sum_{j=0}^{n} \tau_{ij} x_i^j \tag{2.17}$$

where $\tau \in \mathbb{R}^{m \times (n+1)}$ is a matrix of coefficients that parameterise f. Thus, by optimising $g(\tau)$ in Equation (2.16) a function that models the underlying data is constructed.

2.1.4 Unsupervised Learning

Unsupervised learning, also known as self-organisation, requires that a suitable model be fitted to observed patterns without *a priori* knowledge about target outputs for those patterns.

A common unsupervised learning problem is clustering [60] where the goal is to partition observations into homogeneous groupings. The patterns in a given group should be most similar to each other while simultaneously being least similar to observations in other groups, stated formally:

$$\textit{Given}: \quad P = \{\mathbf{p}_t \mid \mathbf{p}_t \in \mathbb{S}\}, \ t \in \{\mathbb{Z} \mid 1 \leq t \leq m\}$$

$$\begin{aligned} \text{find} \quad & C_i \subset P,\ \bigcup C_i = P,\ C_i \cap C_j = \emptyset,\ i, j \in \{\mathbb{Z} \mid 1 \le i, j \le k,\ i \ne j\} \\ \text{such that} \quad & \mathbf{p}_t \in C_i \Longleftrightarrow \sum_{\mathbf{p}_\alpha \in C_i} d(\mathbf{p}_t,\ \mathbf{p}_\alpha) \le \sum_{\mathbf{p}_\beta \in C_j} d(\mathbf{p}_t,\ \mathbf{p}_\beta) \end{aligned} \tag{2.18}$$

where $d(\mathbf{x}, \mathbf{y})$ is a suitable distance metric that measures the dissimilarity between $\mathbf{x}$ and $\mathbf{y}$. The k clusters, C_i, are subsets of the set of patterns, P, such that the observations in a given cluster are related by having similar characteristics. If the clusters are pairwise disjoint then the clustering is a true partition. Equation (2.18) only permits such partitions, however, in general it is possible for a given pattern to belong to multiple clusters, with some degree of membership (refer to Section 2.5.1), yielding a fuzzy clustering. The domain, $\mathbb{S}$, of the m input patterns in P can be anything for which a distance metric can be constructed. If $\mathbb{S} = \mathbb{R}^n$ then a suitable Minkowski metric [7] may be used:

$$d_p(\mathbf{x}, \mathbf{y}) = (\sum_{k=1}^{n} |x_k - y_k|^p)^{\frac{1}{p}} \tag{2.19}$$

for some specified value for p where d_1 and d_2 are the well known Manhattan and Euclidean distances respectively.

The number of clusters inherent to a given data set is generally not known. Choosing a value for k that is either too large or too small is analogous, respectively, to over-fitting and under-fitting in supervised learning.

Missing attributes for patterns can be predicted based on related observations in the same cluster. Appropriate clusters for these patterns are determined using the remaining attributes. An over-fitted model which groups related patterns into separate clusters will be unable to accurately predict missing attributes. Similarly, an under-fitted partitioning that groups unrelated patterns into the same cluster will also have poor prediction ability.

Hierarchical clustering, depicted in Figure 2.4, provides a selection of clusterings where each level in the hierarchy roughly corresponds to a different choice for the value of k. Agglomerative clustering is a bottom up approach where each observation is initially assigned to its own cluster. The closest two clusters are then repeatedly merged until all the observations fall into the same cluster at the root of the tree.

Various strategies exist for determining the merging criteria for clusters. Complete linkage clustering utilises the maximum distance between observations in each cluster. If the minimum distance is used instead then the strategy is known as single linkage clustering. An average linkage clustering results when the mean distance between ob-

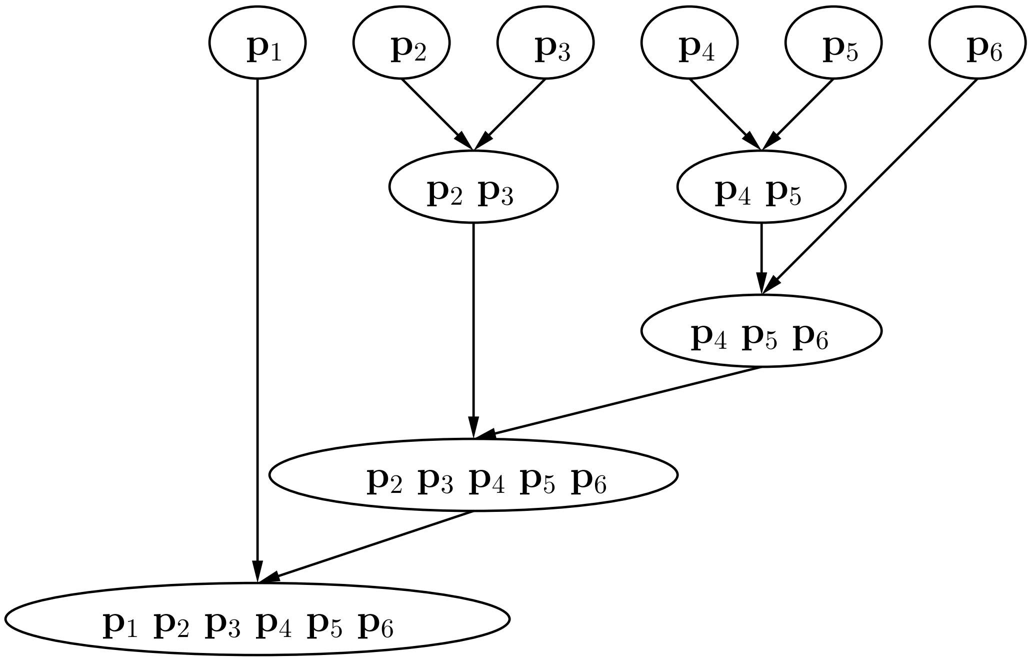
$\mathbf{p}_1$
$\mathbf{p}_2$
$\mathbf{p}_3$
$\mathbf{p}_4$
$\mathbf{p}_5$
$\mathbf{p}_6$
$\mathbf{p}_2\ \mathbf{p}_3$
$\mathbf{p}_4\ \mathbf{p}_5$
$\mathbf{p}_4\ \mathbf{p}_5\ \mathbf{p}_6$
$\mathbf{p}_2\ \mathbf{p}_3\ \mathbf{p}_4\ \mathbf{p}_5\ \mathbf{p}_6$
$\mathbf{p}_1\ \mathbf{p}_2\ \mathbf{p}_3\ \mathbf{p}_4\ \mathbf{p}_5\ \mathbf{p}_6$

Figure 2.4: Hierarchical Clustering

servations of each cluster is used as a criterion. The average linkage distance between cluster $\mathcal{A}$ and cluster $\mathcal{B}$ is defined as:

$$D(\mathcal{A},\mathcal{B}) = \frac{1}{card(\mathcal{A})card(\mathcal{B})}\sum_{x\in\mathcal{A}}\sum_{y\in\mathcal{B}} d(x,y) \tag{2.20}$$

where $card(X)$ is the cardinality of cluster X. Metrics based on intra cluster variance or change in variance (Ward's criterion) are also possible [5].

The clustering problem can be represented by a constrained optimisation problem for a given value of k by determining the optimal assignment vector that maps observations to cluster indexes. One such strategy minimises the distance between observations and the centroids of their clusters, stated formally:

$$\begin{aligned} \textit{Given}: \quad & f(\mathbf{x}) = \sum_{t=1}^{m} d(\mathbf{p}_t, \mathbf{c}_{x_t}),\ i \in \{\mathbb{Z} \mid 1 \le i \le m\} \\ \text{find} \quad & \mathbf{x}^* \in \mathbb{Z}^m \text{ for which } f(\mathbf{x}^*) \le f(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{Z}^m \\ \text{subject to} \quad & -x_i + 1 \le 0 \text{ and } x_i - k \le 0 \end{aligned} \tag{2.21}$$

where $\mathbf{c}_j$ is the centroid of cluster C_j and $\mathbf{x} \in \{\mathbb{Z}^n \mid 1 \le x_i \le k\}$ is the assignment vector such that $x_t = j \iff \mathbf{p}_t \in C_j$.

Clusters defined by a single centroid vector permit only round cluster boundaries. Arbitrarily shaped boundaries can be constructed using a technique known as mixture modelling where each cluster is defined by a weighted density model of different distributions [14].

2.2 Neural Networks

The human brain and nervous system are comprised of billions of nerve cells known as neurons. Each biological neuron is a single cell with receptors called dendrites and an effector called an axon. Neurons are arranged into networks so that the axon of any given neuron can stimulate dendrites of other neurons. When a neuron receives sufficient input stimulus via its dendrites, it fires a signal along its axon which in turn further stimulates the dendrites of other neurons. The arrangement of these relatively simple cells into complex networks generally enables intelligent behaviour in people.

In a similar fashion, the fundamental building block of neural networks in CI is the artificial neuron. By combining these neurons into more complex structures both supervised and unsupervised learning problems can be solved. The canonical feed forward neural network, used for supervised learning, is presented in Section 2.2.1. Other supervised network architectures are mentioned in Section 2.2.2. Unsupervised neural networks such as the learning vector quantiser and self organising feature maps are discussed in Sections 2.2.3 and 2.2.4 respectively.

2.2.1 Feed Forward Neural Networks

Feed forward neural networks can be used to represent nonlinear multivariate relationships [31, 88]. Figure 2.5 illustrates a fully connected three layer network. The layers consist of neurons which compute a function of their inputs and pass the result to the neurons in the following layer. In this manner, the input signal is fed forward from left to right through the network.

The output of a given neuron is characterised by a nonlinear activation function, a weighted combination of the incoming signals, and a threshold value. The threshold can be replaced by augmenting the weight vector to include the input from a constant bias unit. By varying the weight values of the links, the overall function which the network

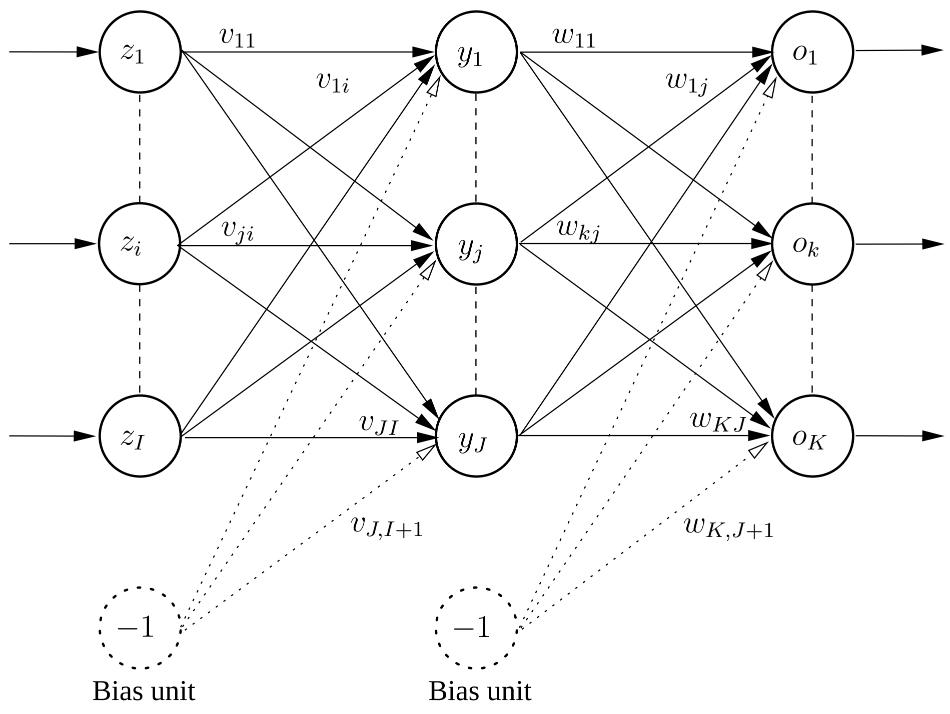

Figure 2.5: Three Layer Feed Forward Neural Network

realises is altered.

The activation signal, o_k for the k^{th} output neuron, for a network with I input, J hidden and K output neurons is given by:

$$o_k = f_{o_k}(\sum_{j=1}^{J+1} w_{kj} y_j) \tag{2.22}$$

$$= f_{o_k}\Big(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I+1} v_{ji} z_i)\Big) \tag{2.23}$$

where v_{ji} and w_{kj} are weights connecting neurons in their respective layers, y_j is the activation signal of the j^{th} hidden neuron, and z_i is the i^{th} input signal. The activation functions f_{y_j} and f_{o_k} are typically the sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.24}$$

which forces outputs into the range $(0, 1)$. Thus, a feed forward network having I inputs, K outputs and sigmoid activation functions realises a nonlinear mapping of the form

$\mathbb{R}^I \rightarrow (0,1)^K$ which is parameterised by the weights v_{ji} and w_{kj}. Alternative activation functions are mentioned in Section 2.2.2.

Training involves finding values for the weights so that the network best approximates the function for a given supervised learning problem (refer to Equation (2.15)). Since the network can only realise values in the range $(0,1)$, target values must be scaled appropriately. In addition, inputs should also be scaled to fall within the active region of the activation functions which, in the case of sigmoid activations, is roughly $[-\sqrt{3}, \sqrt{3}]$. Classification problems are encoded by dedicating a separate output to each label, so that each output represents the posterior probability that an observation belongs to the class associated with that output.

Algorithm 1 Neural Network Back-propagation Training

```
1: Initialise $v_{ji}, w_{kj} \sim U(-1,1)$
2: $t \leftarrow 0$
3: **repeat**
4:   **for all** training patterns **do**
5:     $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}(t) + \alpha \Delta w_{kj}(t-1)$ (refer to Equation (2.25))
6:     $v_{ji} \leftarrow v_{ji} + \Delta v_{ji}(t) + \alpha \Delta v_{ji}(t-1)$ (refer to Equation (2.26))
7:   **end for**
8:   $t \leftarrow t+1$
9: **until** stopping condition
```

Pseudocode for back-propagation learning using gradient descent is presented as Algorithm 1 [116]. Weights are uniformly initialised to small random values and are iteratively updated for each pattern until some stopping criterion is met. The change in output layer weights, derived from the derivative of the SSE over the network, is given by:

$$\Delta w_{kj} = \eta(t_k - o_k)(1 - o_k)o_k y_j \tag{2.25}$$

and the change in hidden layer weights is propagated back using:

$$\Delta v_{kj} = z_i \sum_{k=1}^{K} (1 - y_j) w_{kj} \Delta w_{kj} \tag{2.26}$$

where t_k is the target for the k^{th} output neuron and η is the learning rate. A momentum term which preserves the velocity of weight updates is specified by α.

Instead of simple gradient descent, scaled conjugate gradient techniques [10] or indeed almost any optimisation process could be used to determine appropriate weight values.

2.2.2 Different Network Architectures

There are many ways in which supervised neural network architectures can be customised. Although the number of input and output neurons is defined by the problem, the number of hidden neurons can be varied. At the individual neuron level, different activation functions and methods by which input signals are combined can be utilised. Finally, the network topology can be altered implicitly through dynamic growing, pruning and regularisation; or explicitly at design time as is the case for recurrent and time delay neural networks [31].

Varying the number of hidden neurons affects the complexity of mappings that can be realised by a given neural network. A network with more weights and neurons has more expressive power than one having fewer degrees of freedom. Increasing the number of hidden neurons, however, may lead to over-fitting, since the network would be able to fit inherent noise more easily. Training time is also increased, since more weight updates are required.

In order to fit arbitrary data without over-fitting, the simplest network possible is desired. Regularisation [46, 118] involves driving network weights to zero, in effect removing links to alter the topology, by adding a penalty term to the network error surface that penalises network complexity. Other approaches involve growing or pruning the network by adding or removing neurons respectively when certain triggering criteria are met [31].

Product unit networks [27] utilise higher order combinations of inputs and as such can realise more complex functions with fewer neurons than ordinary summation unit networks. The drawback of a product unit network is that many local minima exist in the error surface causing gradient descent based training algorithms to become trapped at suboptimal solutions more easily. Functional link networks [43] make higher order functions of the inputs available to the hidden layer in an attempt to realise more complex functions with standard summation units.

Sigmoid activation functions are the most common, however, other functions may be used instead. The type of problems for which supervised networks are used typically exhibit nonlinear behaviour. Linear activation functions may be better suited for linearly related data, but will perform poorly for nonlinear relationships. Step functions model binary characteristics in data while ramp functions can realise a mixture between binary and linear relationships. The hyperbolic tangent has a range of $(-1, 1)$, making it suitable for use in hidden layers, since its output nominally falls within the active input region of typical activation functions. The training process should, however, cause weights to be chosen such that inputs lie in the active region irrespective of the output from the previous layer. Although any conceivable activation function may be used, including Gaussians, there is by definition of supervised learning no *a priori* knowledge about the relationship between inputs and targets. As long as their is no good reason to favour one activation function over another, the relative simplicity of the sigmoid makes it most suitable. A combination of sigmoids in the hidden layer and linear output units has also proven to be a good choice [14].

Various network topologies that attempt to model temporal characteristics in data are also possible [54]. Recurrent neural networks attempt to model these temporal characteristics by storing the signal from the hidden or output layers and feeding it back as additional inputs for subsequent training patterns. In a similar fashion, time delay networks maintain the inputs from previous passes as additional inputs to the network.

2.2.3 Learning Vector Quantiser

The Learning Vector Quantiser (LVQ), shown in Figure 2.6, is a two layer unsupervised learning neural network [66]. The input layer has direct connections to the output neurons and there are no bias units. Unlike supervised networks, the weights in an LVQ network have a special meaning. The k^{th} output neuron, o_k, represents a cluster with an I-dimensional centroid comprising the incoming weights, v_{ki}.

Algorithm 2 outlines the training procedure for an LVQ network. As is the case for supervised networks, the weights are initialised to small uniform random values and training patterns are repeatedly presented to the network causing changes to the weight values.

The weights of the nearest output neurons to a given pattern are updated according to the following equation:

$$\Delta v_{ki}(t) = \eta(t)[z_i - v_{ki}(t-1)] \tag{2.27}$$

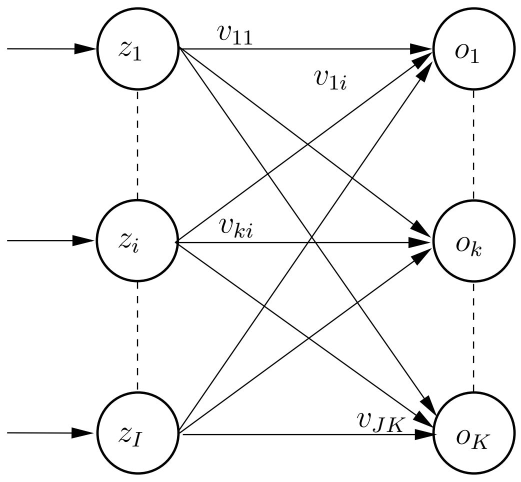

Figure 2.6: Two Layer Learning Vector Quantiser

where $\eta(t)$ is a decaying learning rate so that $\eta(t) \to 0$ as $t \to \infty$. The closest output neuron is determined using the Euclidean distance between the training pattern, $\mathbf{z} \in \mathbb{R}^I$ and the weight vector, $\mathbf{v}_k$, that corresponds to o_k. The set $\kappa_k(t)$ consists of output neuron indices considered to be in the neighbourhood of o_k at time t. The neighbourhood, like the learning rate, is also reduced over time so that $\kappa_j(t) \to \{j\}$ as $t \to \infty$. In addition to the absolute winner j, in terms of closest output neuron, the weights of all the neurons in $\kappa_j(t)$ are typically also updated. A conscience factor can be incorporated into the distance metric in line 5 to penalise output neurons that overly dominate during training [31]. The result is that cluster centroids, represented by the weights of their respective output neurons, are moved towards the most appropriate input patterns.

2.2.4 Self Organising Feature Maps

Conceptually, a Self-Organising Feature Map (SOFM) [66] functions similarly to an LVQ. In fact, the training algorithm is virtually identical. The most notable difference is that the output layer is a two-dimensional map as shown in Figure 2.7. One of the key benefits of SOFMs over LVQ is that the topology of the input space is preserved in the map. That is, if two patterns are closely related in the input space then they usually map to output neurons that are close to each other in terms of coordinate indices in the map. Thus, SOFMs project an I-dimensional input space onto a two-dimensional map space making them a useful data visualisation tool [31].

Algorithm 2 Learning Vector Quantiser Training

```
1: Initialise $v_{ki} \sim U(-1,1)$
2: $t \leftarrow 0$
3: repeat
4:   for all training patterns do
5:     Find $j$ for which $d_2(\mathbf{z}, \mathbf{v}_j)$ is minimised (refer to Equation (2.19))
6:     for all $k \in \kappa_j(t)$ do
7:       $v_{ki} \leftarrow v_{ki} + \Delta v_{ki}(t)$ (refer to Equation (2.27))
8:     end for
9:   end for
10:  $t \leftarrow t + 1$
11: until stopping condition
```

Although SOFM weights may also be initialised to small uniformly distributed random values, there is a better method of performing initialisation that may improve the quality of the mapping [107]. The weights corresponding to the four corners of the map are initialised to the respective four most extreme patterns in the training set. The remaining weights, $\mathbf{v}_{kj}$, are interpolated as follows:

$$\mathbf{v}_{1j} = \frac{\mathbf{v}_{1J} - \mathbf{v}_{11}}{J-1}(j-1) + \mathbf{v}_{11} \tag{2.28}$$

$$\mathbf{v}_{Kj} = \frac{\mathbf{v}_{KJ} - \mathbf{v}_{K1}}{K-1}(j-1) + \mathbf{v}_{K1} \tag{2.29}$$

$$\mathbf{v}_{k1} = \frac{\mathbf{v}_{K1} - \mathbf{v}_{11}}{K-1}(k-1) + \mathbf{v}_{11} \tag{2.30}$$

$$\mathbf{v}_{kJ} = \frac{\mathbf{v}_{KJ} - \mathbf{v}_{1J}}{J-1}(k-1) + \mathbf{v}_{1J} \tag{2.31}$$

$$\mathbf{v}_{kj} = \frac{\mathbf{v}_{kJ} - \mathbf{v}_{k1}}{J-1}(j-1) + \mathbf{v}_{k1} \tag{2.32}$$

for a JxK map with $j \in \{\mathbb{Z} \mid 2 \leq j \leq J-1\}$ and $k \in \{\mathbb{Z} \mid 2 \leq k \leq K-1\}$.

The standard SOFM training algorithm is identical to LVQ except that the weight update for each neuron is now given by:

$$\mathbf{v}_{kj}(t+1) = \mathbf{v}_{kj}(t) + \eta(t)\Phi_{c_{xy},c_{jk}}(t)[\mathbf{z} - \mathbf{v}_{kj}] \tag{2.33}$$

where $\eta(t)$ is once again a decaying learning rate. The coordinates c_{xy} and c_{jk} are the locations of the winning and current neurons respectively on the map. Again, the winning

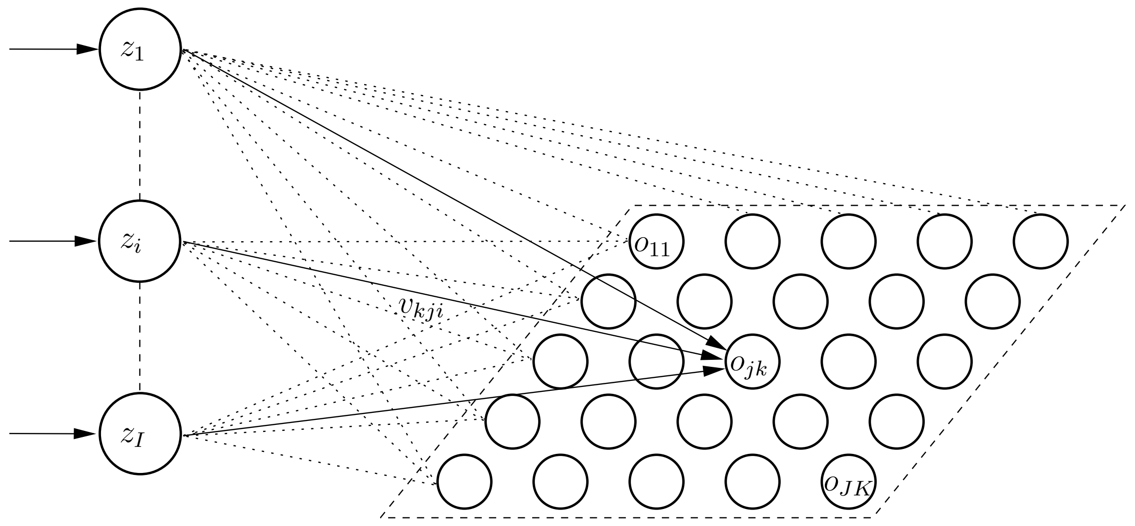

Figure 2.7: 5x5 Self Organising Feature Map

neuron is the one having the closest weight vector, in terms of Euclidean distance, to the current training pattern $\mathbf{z} \in \mathbb{R}^I$. Unlike LVQ, every neuron is typically updated for each training pattern instead of only updating those neurons in an explicit neighbourhood set. The neighbourhood function, $\Phi_{c_{xy},c_{kj}}(t)$, determines the extent which a training pattern has influence over the weights surrounding the winning neuron. Thus, neurons further away from the winning neuron, in map coordinate space, are affected less by a given training pattern. The following Gaussian neighbourhood function is typically used:

$$\Phi_{c_{xy},c_{jk}}(t) = e^{-\frac{||c_{xy}-c_{jk}||_2^2}{2\sigma^2(t)}} \tag{2.34}$$

where $\sigma(t)$ gives the width of the kernel and $\sigma(t) \to 1$ as $t \to \infty$.

A typical SOFM has more output neurons than there are clusters inherent in the training data. Thus, a single output neuron will not, in general, correspond to a single cluster centroid. A unified distance matrix (U-matrix) can be constructed to determine the actual cluster boundaries [31]. The U-matrix is constructed by calculating the distances between each neuron's weight vector and its immediate neighbours in map coordinate space. Large values in the U-matrix are indicative of cluster boundaries while small values indicate groups of neurons belonging to the same cluster. If the map has a high enough resolution then the U-matrix can be plotted as a two-dimensional image that is useful for data visualisation. Figure 2.8 is an example of such a plot with clus-

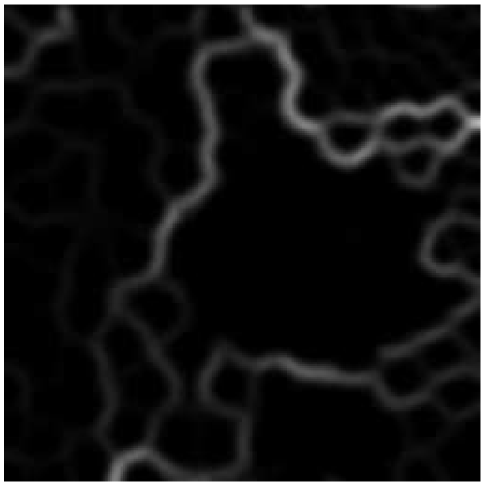

Figure 2.8: Example U-matrix plot

ter boundaries illustrated by white contours that correspond to large U-matrix values. These high resolution maps allow for arbitrary shaped cluster boundaries.

2.3 Evolutionary Computing

All living organisms, ranging from the single celled Amoeba to complex multi-cellular human beings, have a genetic blueprint that describes their physical and behavioural characteristics. This genetic blueprint is made up of DNA (Deoxyribonucleic Acid) arranged into chains of nucleotides called chromosomes. The precise arrangement of the different nucleotides, or genes, defines the characteristics of an organism. The information encapsulated by the DNA is known as the genotype of an organism, while the phenotype is the physical expression of that information. The relationship between genotype and phenotype is typically complex, owing to the influence of pleiotropy and polygeny [77].

Small changes in the genetic material of a population are realised through random mutations and recombination during reproduction between individuals. These changes to the genotype of individuals affect their phenotype and consequently their ability to survive in a given environment. Darwinian theory states that the evolution of a species is guided by competition and natural selection [82]. That is, useful changes in genetic material are preserved from generation to generation, since individuals with better characteristics are the most likely to survive and reproduce.

Algorithm 3 General Evolutionary Computing Framework

```
1: $t \leftarrow 0$
2: $P(t) \leftarrow \text{initialise}(\mu)$
3: $F(t) \leftarrow \text{evaluate}(P(t), \mu)$
4: **repeat**
5:     $P'(t) \leftarrow \text{recombine}(P(t), \Theta_r)$
6:     $P''(t) \leftarrow \text{mutate}(P'(t), \Theta_m)$
7:     $F(t) \leftarrow \text{evaluate}(P''(t), \lambda)$
8:     $P(t+1) \leftarrow \text{select}(P''(t), F(t), \mu, \Theta_s)$
9:     $t \leftarrow t + 1$
10: **until** stopping condition
```

Evolutionary Computing (EC) is strongly based on the principles of natural evolution. A general framework for evolutionary optimisation that encompasses these principles is given in Algorithm 3 [109]. A population of μ individuals is initialised within the search space of an optimisation problem so that $P(t) = \{\mathbf{x}_i(t) \in \mathbb{S} \mid 1 \leq i \leq \mu\}$. The search space $\mathbb{S}$ may be the genotype or phenotype depending on the particular evolutionary approach being utilised. The fitness function f, which is the function being optimised, is used to evaluate the goodness individuals so that $F(t) = \{f(\mathbf{x}_i(t)) \in \mathbb{R} \mid 1 \leq i \leq \mu\}$. Obviously, the fitness function will also need to incorporate the necessary phenotype mapping if the genotype space is being searched.

Searching involves performing recombination of individuals to form offspring, random mutations and selection of the following generation until a solution emerges in the population. The parameters Θ_r, Θ_m and Θ_s are the probabilities of applying the recombination, mutation and selection operators respectively. Recombination involves mixing the characteristics of two or more parents to form offspring in the hope that the best qualities of the parents are preserved. Mutations, in turn, introduce variation into the population thereby widening the search. In general, the recombination and mutation operators may be identity transforms so that it is possible for individuals to survive into the following generation unperturbed. Finally, the λ new or modified individuals are re-evaluated before the selection operator is used to pare the population back down to a size of μ. The selection operator provides evolutionary pressure so that the most fit individuals survive into the next generation. While selection is largely based on the fitness of individuals, it is probabilistic to prevent premature convergence of the population.

Genetic algorithms, which generally search the genotype space, are summarised in the next section. Section 2.3.2 covers a specialisation of genetic algorithms where the genotype is a space of executable program trees. Evolutionary programming, discussed in Section 2.3.3, concentrates on searching the phenotype space. Evolutionary strategies, which dynamically evolve strategy parameters, are discussed in Section 2.3.4. Finally, cultural and co-evolutionary extensions are considered in Sections 2.3.5 and 2.3.6 respectively.

2.3.1 Genetic Algorithms

Genetic Algorithms (GAs) [47] fit neatly into the general EC framework already presented in Algorithm 3. Thus, the only remaining requirement, to fully describe a GA, is the definition of a specific genotype representation along with suitable recombination, mutation and selection operators.

Traditional GAs [56] represent individuals as binary bit strings. Numeric phenotypes are usually encoded using Gray's code in the genotype to reduce pleiotropic variation in the phenotype. That is, the genotypic Hamming distance is minimised for small differences in phenotypic values. A real ($\mathbb{R}$) valued genotype, having an identical phenotype, is also possible, provided that recombination and mutation are suitably defined for real values. In fact, any representation, for which suitable operators can be defined, may be used. For example, genetic programming, presented in the following section, is a special type of GA having a tree based representation.

Reproduction, or the mixing of genetic material, between multiple individuals is known as crossover in the context of GAs. Figure 2.9 illustrates three types of crossover that can be defined for binary coded individuals. Each of them is defined in terms of a binary mask and is able to produce two offspring from a pairing of two parents. The mask determines the parent from which the offspring inherit their genetic material. In the case of uniform crossover, a random mask is generated that results in offspring composed of random components of the two parent's genetic material. For one-point crossover, a random offset in the mask is chosen, so that all components up to that offset are inherited from the one parent and the rest from the other. Similarly, for two-point

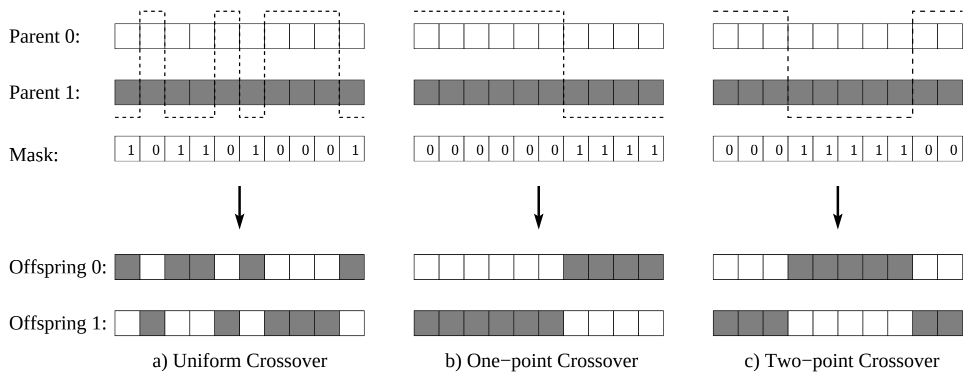

Figure 2.9: Crossover Operators

crossover, there are two offsets chosen so that only the components between the two positions are inherited from the one parent. For real valued genes, arithmetic crossover may be defined for two individuals $\mathbf{x}_a$ and $\mathbf{x}_b$ as follows:

$$\mathbf{x}_a(t+1) = \rho\mathbf{x}_a(t) + (1-\rho)\mathbf{x}_b(t) \tag{2.35}$$

$$\mathbf{x}_b(t+1) = \rho\mathbf{x}_b(t) + (1-\rho)\mathbf{x}_a(t) \tag{2.36}$$

where $\rho \sim U(0,1)$ is a uniform random variate.

Mutation is typically performed with a fairly low probability, since existing good solutions may be disturbed if the mutation rate is too high. A suitable mutation operator for binary coded individuals inverts bits subject to a given probability, while real valued mutation can be achieved by adding Gaussian noise.

An elitism operator is usually implemented to select a few good individuals, the elite, to survive into the following generation. This can be achieved trivially, by adding the new and modified individuals, obtained through recombination and mutation, to the existing population and subjecting the entire pool to selection.

Various selection strategies exist, including tournament, proportional, and rank-based selection [31]. Tournament selection involves repeatedly selecting k individuals randomly from the population and then selecting the individual with the best fitness out of that group. A proportional strategy selects individuals in proportion to their fitness by sampling the following distribution:

$$P(\mathbf{x}_i(t)) = \frac{f(\mathbf{x}_i(t))}{\sum_{n=1}^{\mu} f(\mathbf{x}_n(t))} \tag{2.37}$$

so that $P(\mathbf{x}_i(t))$ is the probability of selecting the i^{th} individual from the population at time t. Finally, rank-based selection techniques sample the rank ordered distribution of individuals instead of considering absolute fitness values.

2.3.2 Genetic Programming

Any algebraic expression can be trivially represented in tree form. Non-terminal tree nodes represent mathematical operators so that their children correspond with the parameters of the operator in question. Variables and constants, in turn, are represented as terminal nodes in the tree. Figure 2.10 is an example tree for the expression $\sin(\frac{p}{q})(\log(r) - e^{s+1.5})$. In a similar fashion, a parse tree, for arbitrary computer programmes in any language, can be constructed.

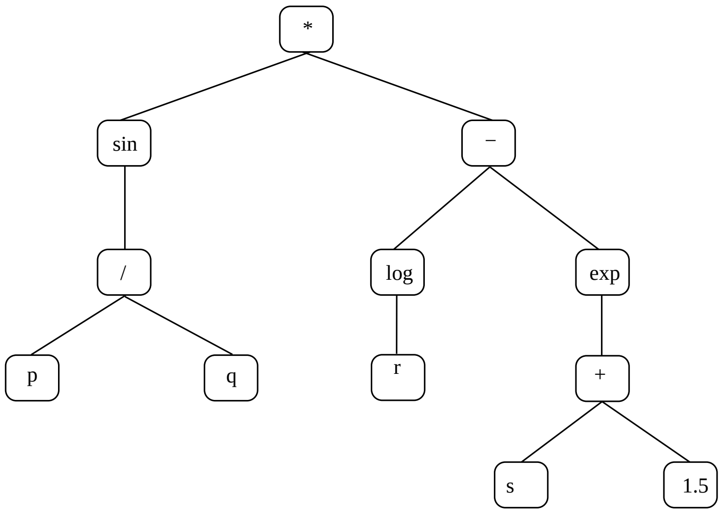

Figure 2.10: Genetic Program Tree Representation

Genetic programmes are nothing more than GAs, with the genotype being parse trees for executable programmes in a given language [67]. Consequently, the phenotype is the behaviour of those programmes at execution time. The fitness function is a measure of how well a programme performs a specified task. Selection is also analogous to GAs, so all that remains is to define suitable crossover and mutation operators for tree structures.

Crossover is trivial, a random node in each parent tree is selected. These two nodes, along with their descendents, are swapped, forming two possible offspring. That is, the selected subtree of one parent is replaced with the selected subtree of the other.

Several mutation operators, which should be used together, can be defined [31]:

- **Function node mutation:** A randomly selected non-terminal node has its operator replaced with another operator that has the same cardinality.
- **Terminal node mutation:** A randomly selected terminal node is replaced with another valid terminal node.
- **Swap mutation:** A non-terminal node, having more than one child, is selected and order of its children are altered.
- **Grow mutation:** A randomly selected node is replaced with a randomly generated subtree that has a predetermined maximum depth.
- **Gaussian mutation:** A terminal node which represents a constant is randomly selected and mutated by adding Gaussian noise.
- **Trunc mutation:** A randomly selected non-terminal node is replaced with a valid terminal node.

2.3.3 Evolutionary Programming

Evolutionary Programming (EP) [36, 37] can be classified in the EC framework in Algorithm 3 by leaving out the fifth step, or equivalently, defining recombination as an identity transform. That is, EP relies solely on mutation and does not make use of any recombination. In addition, EP does not explicitly distinguish between genotype and phenotype. Rather, mutations are defined based on the problem domain, implicitly making EP a phenotypic optimisation process.

EP was originally developed to evolve finite-state machines by defining the following mutations: change an output symbol; change a state transition; add a state; delete a state; or change the initial state. Real valued domains can make use of Gaussian mutation, as is the case for real valued genotypes in GAs. In any event, the mutation operator used will be problem specific, since EP performs a search of the phenotype. Mutation should be biased towards making small changes but should allow for large mutations, particularly early on in the search, to enable the optimisation process to avoid local extrema.

2.3.4 Evolutionary Strategies

The general EC framework defined in Algorithm 3 has many parameters that may affect its performance in various ways. In the context of Evolutionary Strategies (ES) [93, 94], these are known as strategy parameters. The primary principle of ES is to concurrently evolve these strategy parameters alongside the solution to the problem under optimisation. In this way, ES are able to more optimally adapt their strategy to the problem at hand.

Like other EC paradigms, implementations of ES also define their own representation as well as recombination, mutation and selection operators. Canonical ES specify mutation and crossover operators defined for vectors of real values, inherently making ES a phenotypic search process. Thus, the standard representation for ES is a real valued solution vector augmented by one or more strategy parameters so that:

$$\mathbf{x}(t) \in \{(\mathbb{R}^n, \mathbb{R}^s)\} \tag{2.38}$$

for an individual $\mathbf{x}(t)$ of solution dimension n with s strategy parameters. It is possible, however, to apply similar strategy parameters to genotypic search algorithms to enhance their performance. In general, any parameter that influences the evolutionary process can be appended to an individual's representation. Individuals that are performing poorly may have their strategy parameters adjusted more dramatically under the assumption that their poor performance is due to a bad choice of strategy.

Specifically, mutation is enhanced by associating additional parameters with each individual. The simplest of these schemes associates a standard deviation, $\sigma(t)$, with each member of the population so that the mutation operator perturbs the solution vector as follows:

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \sigma(t+1)\xi \tag{2.39}$$

where $\xi \in \mathbb{R}^n$ with each $\xi_i \sim N(0,1)$ a normally distributed random variate, while the standard deviation for each successive generation is updated according to:

$$\sigma(t+1) = \sigma(t)^{e^{\rho\sqrt{n}}} \tag{2.40}$$

where $\rho \sim N(0,1)$. More elaborate schemes that include a standard deviation along with a matrix of rotation angles have also been devised [31].

Crossover can be applied to both the solution vector and the strategy parameters. ES define different crossover operators to standard GAs. Local crossover resembles uniform crossover in that an offspring is created by selecting random components from two parents. Global crossover, however, selects random components from the entire population to generate a single offspring. In addition to simply selecting random components, arithmetic crossover or simple averaging can be performed between multiple parents.

Two primary selection strategies have been defined for ES. The first, known as $(\mu+\lambda)$, selects successive generations from the combination of the previous generation and all the offspring. The second, known as (μ, λ), selects the following generation from the set of offspring only. The former implicitly implements a form of elitism operator while the latter does not allow for individuals to survive through successive generations and requires that $1 \leq \mu < \lambda < \infty$.

2.3.5 Cultural Evolution

Cultural evolution [96] is based on the premise that cultural properties in a population evolve at a faster rate than genetic properties. The search process is biased by a cultural belief space that focuses the search in areas that the population believes contains good solutions. This belief space, which stores the best behavioural traits of the population over time, is used to enhance and accelerate the search process.

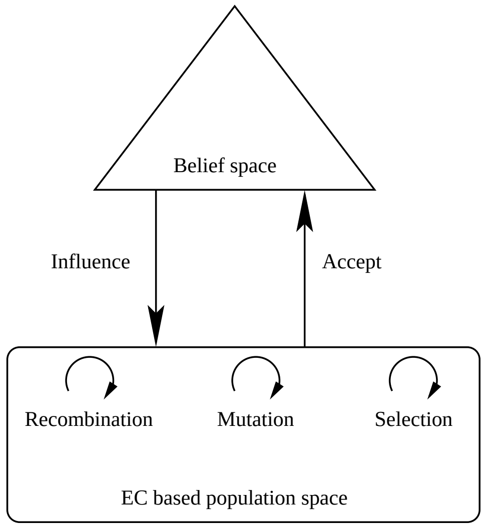

Figure 2.11: Cultural Algorithm

Cultural algorithms deviate from the model given in Algorithm 3 by maintaining two separate search spaces. The first, the population space, is an instance of one of the already mentioned EC algorithms, perhaps a GA or an EP algorithm. Secondly, the belief space serves as a repository of knowledge gained by the main population during the entire search process. Figure 2.11 illustrates the relationship between these two spaces. An acceptance function specifies how this knowledge is communicated from the main population and incorporated into the belief space. An influence function, in turn, determines how the search process of the main population is influenced by the knowledge in the belief space.

The choice of functions that govern acceptance of knowledge into the belief space and the influence of that knowledge on the population are problem specific. In the case of $\mathbb{R}^n$ domains, the belief space may be defined by the intervals in which the solution is believed to exist in each dimension. Thus, the acceptance function is defined as the bounding hyper-rectangle created by a given percentage of the best performing individuals in the population. Influence of the population is achieved through a modified mutation operator. Individuals lying further outside the range defined by the belief space are subjected to larger mutation step sizes while those within the range are mutated by a smaller amount. In this way, individuals are encouraged to search the belief space more thoroughly. Constrained optimisation can also be supported by forcing the conformance of belief space to those constraints.

2.3.6 Coevolution

Coevolution is an extension of EC into multiple competing or cooperating populations which work together to solve a given problem. The fitness of a given individual becomes a subjective measure relative to the other populations being co-evolved.

For cooperating populations, the solution vector may be split into smaller dimensions with each subpopulation solving only the part of the vector for which it is responsible [117]. In this case, fitness must be measured within the context of the other populations since the objective function requires a full length solution vector to be calculated. Alternatively, the search space itself may be partitioned into intervals, or a global "black board" may be used for sharing partial solutions between populations.

In the case of competing populations, a key benefit is that an absolute fitness measure is not a requirement. The fitness of an individual in one population is measured relative to the performance of individuals in competing populations by playing the individuals against one another [55].

Various sampling strategies for selecting the individuals from other populations that take part in the relative fitness evaluation exist [31]:

- **All versus all:** The fitness for a given individual is calculated relative to all the individuals in other populations.
- **Random:** Fitness is calculated relative to a random group of individuals selected from the other populations.
- **Tournament:** The best individual within a random subgroup of the other populations is selected and fitness is calculated relative to this individual..
- **All versus best:** Fitness is calculated relative to the best performing individual in other populations.

2.4 Swarm Intelligence

Swarm Intelligence models the naturally observed phenomenon of a population, or swarm, of relatively unsophisticated organisms, through their social interactions, to be able to realise globally intelligent behavioural patterns. An example of this phenomenon is the ability of ants to find the most optimal routes to food sources. The individual ants themselves are very simple creatures lacking the ability to think or reason, yet as a colony, they appear able to perform the complex task of determining the optimal routes to food.

Like the EC paradigm discussed in Section 2.3, swarm intelligence approaches are also population based, however, that is where the similarity ends. EC is primarily concerned with evolutionary operators, such as mutation and recombination, to bring about variation in a population, and selection, as a means to focus the search into areas that promise the best results. Swarm intelligence, on the other hand, concentrates on modelling the social interactions between individuals in a population, which usually have a specific task to perform, and typically does not exhibit any kind of selection pressure that governs the survivability of particular individuals.

Particle swarm optimisation, discussed in the following section, exchanges experiential knowledge about the search surface between particles as a means of social interaction. Section 2.4.2 overviews ant systems where interaction between individuals occurs indirectly by means of modifications to the environment in which they function. By modelling these social interactions useful algorithms have been devised for solving numerous problems including function and route optimisation as well as unsupervised clustering.

2.4.1 Particle Swarm Optimisation

Particle swarm optimisation [63, 28] was originally inspired by the flocking behaviour of birds. In terms of this bird flocking analogy, a particle swarm optimiser consists of a number of particles, or birds, that fly around a search space, or the sky, in search of the best location. Each of these particles corresponds to a simple agent that moves through a multi-dimensional search space sampling an objective function at various positions. The motion of a given particle is dictated by its velocity which is continuously updated in order to pull it towards its own best position and the best positions experienced by the rest of the swarm. This behaviour ultimately results in an optimiser that converges to good solutions of an objective function of the form $f : \mathbb{R}^n \to \mathbb{R}$.

The velocity update for each dimension, given by the subscript $j \in \{\mathbb{Z} \mid 1 \leq j \leq n\}$, of the i^{th} particle with position $\mathbf{x}_i(t) \in \mathbb{R}^n$ and velocity $\mathbf{v}_i(t) \in \mathbb{R}^n$ at time t is given by the following equation [63, 28, 100]:

$$v_{i,j}(t+1) = wv_{i,j}(t) \; + c_1 r_{1,j}(y_{i,j}(t) - x_{i,j}(t)) \; + c_2 r_{2,j}(\hat{y}_{i,j}(t) - x_{i,j}(t)) \tag{2.41}$$

where $w \in \{\mathbb{R} \mid 0 \leq w < 1\}$ is an inertia weight that preserves some of the previous velocity; $c1$ and $c2 \in \{\mathbb{R} \mid 0 \leq c_1, c_2 \leq 2\}$ are acceleration coefficients; and $r_{1,j}, r_{2,j} \sim U(0,1)$ are drawn from two independent uniform random distributions. The vector $\mathbf{y}_i(t) \in \mathbb{R}^n$ is the best position found by the individual particle, while $\hat{\mathbf{y}}_i(t) \in \mathbb{R}^n$ represents the best position found by other particles in the swarm. Various neighbourhood strategies determine which particles participate in the social network of a given particle, so that $\hat{\mathbf{y}}_i(t)$ represents the best solution found by the particles in the neighbourhood of the i^{th} particle.

The second term in Equation (2.41) is known as the cognitive component, since it takes into account a particle's own experience of the search terrain. Setting $c_2 \leftarrow 0$

results in a cognition only optimiser having no social interaction between the particles. Conversely, setting $c_1 \leftarrow 0$ leaves only the social component, the third term in the equation. The acceleration coefficients can be chosen (or varied over time) to prioritise the influence of a particle's own cognition or its social interaction with the rest of the swarm. Whenever:

$$\frac{c_1 + c_2}{2} - 1 < w \tag{2.42}$$

holds, particles will exhibit convergent trajectories, otherwise they will not stabilise [113]. Alternatively, a $V_{\max}$ strategy can be used to reduce the likelihood of divergence by enforcing an upper bound on particle velocities.

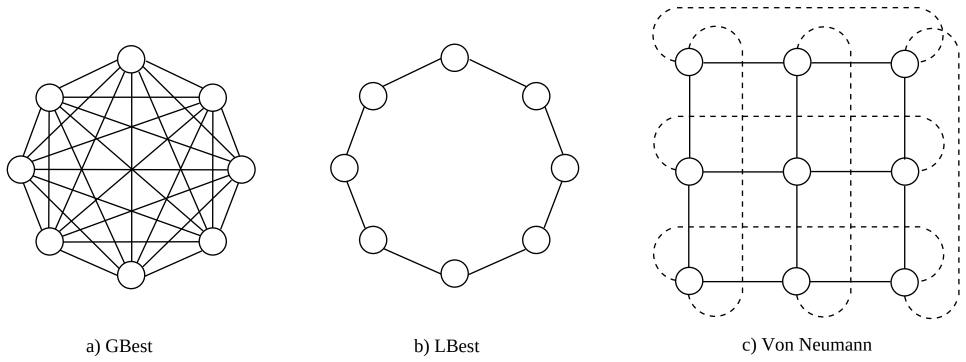

Figure 2.12: Typical Neighbourhood Topologies

The influence of various neighbourhood topologies on the PSO has been been studied extensively [29, 101, 61, 64, 108, 90]. Figure 2.12 illustrates the best known neighbourhood topologies. The GBest, or global best, topology includes every particle of the swarm within the social network of every other particle. LBest, or local best, only considers a particle's immediate neighbours, in terms of particle index, to be socially connected. Finally, the Von Neumann architecture, taking the form of a grid with wrap-around, considers the particles above, below, to the left and to the right to be within a given particle's neighbourhood. The more densely connected the neighbourhood, the quicker information about good solutions is communicated amongst particles in the swarm. Neighbourhood topologies such as LBest and Von Neumann result in superior solutions at the cost of slower convergence, since diversity within the swarm is maintained longer.

Algorithm 4 outlines the Particle Swarm Optimiser (PSO). Initialisation is performed by randomly placing the particles within the search space. All velocities are initialised

Algorithm 4 Particle Swarm Optimiser

```
1: for all particles $i$ do
2:     Initialise $x_{i,j}(0) \sim U(x_{min,j}, x_{max,j})$
3:     $\mathbf{y}_i(0) \leftarrow \mathbf{x}_i(0)$
4:     $\hat{\mathbf{y}}_i(0) \leftarrow \mathbf{x}_i(0)$
5:     $\mathbf{v}_i(0) \leftarrow \mathbf{0}$
6: end for
7: $t \leftarrow 0$
8: repeat
9:     for all particles $i$ do
10:        if $f(\mathbf{x}_i(t)) > f(\mathbf{y}_i(t))$ then
11:            $\mathbf{y}_i(t) \leftarrow \mathbf{x}_i(t)$
12:            if $f(\mathbf{x}_i(t)) > f(\hat{\mathbf{y}}_i(t))$ then
13:                $\hat{\mathbf{y}}_i(t) \leftarrow \mathbf{x}_i(t)$
14:            end if
15:        end if
16:        Update $\mathbf{v}_i(t+1)$ according to Equation (2.41)
17:        $\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1)$
18:    end for
19:    $t \leftarrow t+1$
20: until stopping condition
```

to zero and the personal best positions of the particles are their initial positions. Steps 10 through 15 maintain the personal best positions, $\mathbf{y}_i(t)$, as well as the neighbourhood best position, $\hat{\mathbf{y}}_i(t)$, where the fitness function is given by f. Thus, the particle positions are moved, in step 17, towards their own best positions and the best positions found by the swarm according to Equation (2.41). Upon termination, the best solution found to the optimisation problem is given by the position of the particle with the best fitness.

2.4.2 Ant Systems

Artificial ant systems model the social interaction and seemingly intelligent behaviour of naturally occurring colonies of ants. These social interactions are due to a phenomenon known as stigmergy, characterised by a lack of centralised control and indirect communication by means of modifications to the environment. The emergent behaviour of the colony is observed in their ability to, amongst others, locate optimal food resources and perform nest brooming, including cemetery maintenance [31].

This section describes an optimisation algorithm, applicable to the TSP discussed in Section 2.1.2, followed by an algorithm for performing unsupervised clustering. The former models the way ants optimise paths to food sources, and the latter is based on their cemetery maintenance behaviour.

Ant Colony Optimisation

Foraging in ant colonies is governed by pheromone deposits along paths to food. In general, pheromones are invisible chemicals secreted by organisms which, when detected by the senses, cause an instinctual reaction in another organism. In particular, foraging ants tend to follow paths with higher concentrations of pheromone deposits.

Pheromones are deposited along a given path by the ants that traversed that path at an earlier time. The pheromone following nature of ants combined with the fact that pheromone deposits evaporate over time, results in the shortest paths containing the highest pheromone concentrations. This is because an ant that discovers a shorter path will return sooner, depositing more pheromones, on the way to a food source and again on the way back, as well as more recent pheromones than an ant on a longer path. As more and more ants start to follow the shorter path, due to a higher pheromone concentration, a positive feedback loop is created until virtually all the ants follow the shortest path. Thus, social interaction and coordination for foraging occurs indirectly through pheromone deposits which modify the environment.

Algorithm 5 models the foraging behaviour of ants to solve the TSP (refer to Section 2.1.2) [26]. Each edge of a TSP graph is associated with a pheromone intensity between city i and j at time t denoted by $\tau_{ij}(t)$. The probability, $\Phi_{ij,k}(t)$, for ant k at city i to choose j as the next city to visit is given by:

$$\Phi_{ij,k}(t) = \frac{\tau_{ij}(t)^{\alpha}\eta_{ij}^{\beta}}{\sum_{c \in C_{i,k}} \tau_{ic}(t)^{\alpha}\eta_{ic}^{\beta}} \tag{2.43}$$

where $C_{i,k}$ is the set of city indices that ant k still needs to visit from city i and η_{ij} is the economy of travelling from city i to j. The parameters, α and β, control the respective

Algorithm 5 Ant Colony Optimiser for TSP

```
1: Initialise $\tau_{ij}(0) \sim U(0, max)$
2: Place all ants $k \in \{\mathbb{Z} \mid 1 \leq k \leq m\}$ at origin city
3: Let $T^+$ be the shortest tour, and $L^+$ its length
4: $t \leftarrow 0$
5: repeat
6:   for all ants $k$ do
7:     Build tour $T_k(t)$ by choosing successive cities with probability $\Phi_{ij,k}(t)$
       (refer to Equation (2.43))
8:     Compute length of route, $L_k(t)$
9:     if $L_k(t) < L^+$ then
10:      $T^+ \leftarrow T_k(t)$
11:      $L^+ \leftarrow L_k(t)$
12:    end if
13:  end for
14:  Update pheromone deposits using Equation (2.44)
15:  $t \leftarrow t + 1$
16: until stopping condition
```

importance of pheromone intensities, $\tau_{ij}(t)$, and local cost information, $\eta_{ij} = 1/d_{ij}$, where d_{ij} is a suitable Minkowski distance metric.

The algorithm randomly initialises the pheromone intensities, places a number, m, of ants at the originating city and then proceeds to iteratively build tours, T_k, for each ant k according to Equation (2.43) while continuously maintaining pheromone updates according to:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}(t) \tag{2.44}$$

where ρ is known as a forgetting factor which causes pheromone depletion over time. The net change in pheromone intensity, $\Delta\tau_{ij}(t)$, at time t between city i and j is given by:

$$\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t) \tag{2.45}$$

which is the sum of the deltas over all ants where the contribution of each ant is, in turn, given by:

$$\Delta\tau_{ij,k}(t) = \begin{cases} Q/L_k(t) & \text{if } (i,j) \in T_k(t) \\ 0 & \text{if } (i,j) \notin T_k(t) \end{cases} \tag{2.46}$$

where Q is of the same order of magnitude as the optimal route length and $L_k(t)$ is the length of the tour just taken by ant k. The contribution of an ant to the pheromone intensity between cities i and j is zero if the ant did not traverse that edge during its tour. When the algorithm terminates, the optimal tour found is given by T^+ and its length by L^+.

Ant Colony Clustering

Several species of ants have been observed to cluster corpses into cemeteries in order to tidy their nests. While not much is known about this behaviour, it has provided the inspiration for an algorithmic solution to the unsupervised clustering problem [15].

Algorithm 6 outlines an approach for clustering using a colony of artificial ants. The fundamental idea is to allow ants to roam a grid containing data vectors, picking up those vectors which are dissimilar from their surrounding vectors and dropping them in areas having more similar vectors.

The local density function, $f(\mathbf{z}_i, r)$, which is a measure of the average similarity of the vector $\mathbf{z}_i$ to the vectors in a neighbourhood around the location r is given by:

$$f(\mathbf{z}_i, r) = \frac{1}{s^2} \sum_{\mathbf{z}_j \in \mathcal{N}_{sxs}(r)} [1 - \frac{d(\mathbf{z}_i, \mathbf{z}_j)}{\alpha}] \tag{2.47}$$

where $\mathcal{N}_{sxs}(r)$ is the set of vectors in a square neighbourhood of width s around r and $d(\mathbf{z}_i, \mathbf{z}_j)$ is the dissimilarity, a Minkowski metric, between two vectors $\mathbf{z}_i$ and $\mathbf{z}_j$ with α controlling the scale of the dissimilarity measure.

An unladen ant at location r which is occupied by a vector $\mathbf{z}_i$ picks up that vector with probability:

$$p_p(\mathbf{z}_i, r) = \left(\frac{k_1}{k_1 + f(\mathbf{z}_i, r)}\right)^2 \tag{2.48}$$

where k_1 is a constant which can be used to tune the sensitivity of the resultant probability to $f(\mathbf{z}_i, r)$. Equation (2.48) has the property that vectors which are highly similar to those in their neighbourhood have a low probability of being picked up. Conversely, lower values of $f(\mathbf{z}_i, r)$ result in a high probability of $\mathbf{z}_i$ being picked up, since $p_p(\mathbf{z}_i, r) \to 1$ as $f(\mathbf{z}_i, r) \to 0$.

Algorithm 6 Ant Colony Clustering

```
1:  Place each data vector $\mathbf{z}_i$ randomly on grid
2:  Place all ants $k \in \{\mathbb{Z} \mid 1 \leq k \leq m\}$ randomly on grid
3:  repeat
4:      for all ants $k$ do
5:          Let $r$ be the location of ant $k$
6:          if unladen($k$) and occupied($r$, $\mathbf{z}_i$) then
7:              Compute $f(\mathbf{z}_i, r)$ and $p_p(\mathbf{z}_i, r)$ (refer to Equations (2.47) and (2.48))
8:              if $U(0, 1) \leq p_p(\mathbf{z}_i, r)$ then
9:                  Pick up data vector $\mathbf{z}_i$
10:             end if
11:         else if laden($k$, $\mathbf{z}_i$) and empty($r$) then
12:             Compute $f(\mathbf{z}_i, r)$ and $p_d(\mathbf{z}_i, r)$ (refer to Equations (2.47) and (2.49))
13:             if $U(0, 1) \leq p_d(\mathbf{z}_i, r)$ then
14:                 Drop data vector $\mathbf{z}_i$
15:             end if
16:         end if
17:         Move ant $k$ to randomly selected neighbouring site not occupied by another ant
18:     end for
19: until stopping condition
```

Alternatively, a laden ant carrying a vector $\mathbf{z}_i$ at an unoccupied location r drops its vector with probability:

$$p_d(\mathbf{z}_i, r) = \begin{cases} f(\mathbf{z}_i, r) & \text{if } f(\mathbf{z}_i, r) < k_2 \\ 1 & \text{otherwise} \end{cases} \tag{2.49}$$

where k_2 is a constant that biases towards dropping vectors as k_2 is made smaller, since $p_d(\mathbf{z}_i, r) \to 1$ as $k_2 \to 0$.

An obvious consequence of Algorithm 6 is that the grid must be large enough to accommodate all the data patterns as well as sufficient ants. Strategies that mitigate over-fitting, such as having ants moving at different speeds, can also be implemented [31].

2.5 Fuzzy Systems

Traditional expert systems [45], which typically use first-order predicate calculus to represent rules, rely on boolean logic where an element either belongs to a set or it does not. That is, the law of the excluded middle applies and set membership is precise. Fuzzy inferencing systems, on the other hand, are based on the properties of fuzzy sets [125] where membership is no longer precise. Instead, an element belongs to a given set with an associated degree of membership.

The ability to model the fuzzy, or imprecise, membership of an element to a set enables inferencing based on linguistic terms. Production rules governing a fuzzy controller can be described using words or simple sentences in natural language as opposed to formal predicate calculus statements. This enables a domain expert, who typically would not have an advanced knowledge of first-order predicate logic, to describe the rules that govern a given system using domain specific linguistic terms which may be better understood.

Section 2.5.1 overviews the theory of fuzzy sets and linguistic variables. Fuzzy controllers, discussed in Section 2.5.2, build on this theory to provide a powerful inferencing engine that can be used to solve control problems based on domain knowledge provided by an expert.

2.5.1 Fuzzy Sets

Fuzzy sets [125] are characterised by a membership function of the form:

$$\mu_A : X \to [0, 1] \tag{2.50}$$

where $\mu_A(x)$, $\forall x \in X$, indicates the degree, or certainty, that x belongs to the fuzzy set A, and X is known as the universe of discourse. Traditional boolean set membership can be modelled by a membership function, $\mu_A(x)$, which strictly takes on the values 0 or 1.

Table 2.1 defines fuzzy set theoretic operators that are analogues for their traditional set counterparts. Two fuzzy sets are equivalent if and only if their membership functions are identical. A fuzzy set is a superset of another set if and only if it contains all the elements of the other set to at least the same degree of membership. The complement of a set contains the same elements as the original set, but with complimentary degrees of

Table 2.1: Fuzzy Set Theoretic Operators

Operator	**Definition**
Equality	$A = B \iff \mu_A(x) = \mu_B(x),\ \forall x \in X$
Containment	$A \subset B \iff \mu_A(x) \leq \mu_B(x),\ \forall x \in X$
Complement	$\mu_{\overline{A}}(x) = 1 - \mu_A(x),\ \forall x \in X$
Intersection	$\mu_{A\cap B} = \min\{\mu_A(x), \mu_B(x)\},\ \forall x \in X$, or $\mu_{A\cap B} = \mu_A(x)\mu_B(x),\ \forall x \in X$
Union	$\mu_{A\cup B}(x) = \max\{\mu_A(x), \mu_B(x)\},\ \forall x \in X$, or
	$\mu_{A\cup B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x),\ \forall x \in X$

membership, so that an element having a high degree of membership has a proportionally low degree of membership to the complement. The intersection operator may be defined as the minimum of the degrees of membership of elements to each set, or it may be defined as the product of the membership functions. The product version is the stronger of the two operators, resulting in lower degrees of membership for the intersection. Similarly, the union may be defined in terms of the maximum degree of membership, or it may be defined algebraically. In the limit, a series of unions cumulatively tends to 1 and a series of intersections tends to 0, irrespective of the degrees of memberships to the individual sets.

Linguistic variables and their associated hedges [126, 127, 128] express words and sentences, in natural language, in terms of fuzzy set memberships. Consider as an example, the concept of a person's age as a linguistic variable. The linguistic variable *age* might take on values such as *young*, *middle aged* and *old*. Each of these values defines a fuzzy set, associated with a membership function that models its semantics. Figure 2.13 illustrates three possible membership functions, defined using Gaussians, for the values *young*, *middle aged* and *old* respectively. Further, hedges such as *very*, *fairly*, *somewhat* and *slightly* may be used to modify a membership function.

Numerous hedges may be defined, with the primary types of hedges given by the following equations:

$$\text{Concentrate}: \quad \mu_{A'}(x) = \mu_A(x)^p \tag{2.51}$$

$$\text{Dilate}: \quad \mu_{A'}(x) = \mu_A(x)^{1/p} \tag{2.52}$$

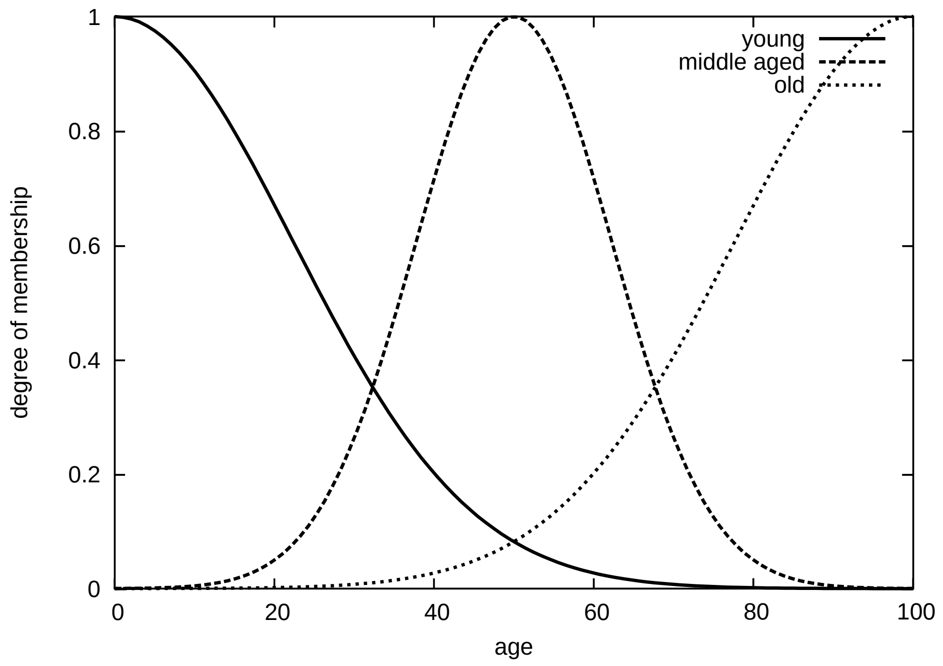

Figure 2.13: Membership Functions for *Age* Linguistic Variable

$$\text{Intensify}: \quad \mu_{A'}(x) = \begin{cases} 2^{p-1}\mu_A(x)^p & \text{if } \mu_A(x) \leq 0.5 \\ 1 - 2^{p-1}(1-\mu_A(x))^p & \text{otherwise} \end{cases} \tag{2.53}$$

$$\text{Blur}: \quad \mu_{A'}(x) = \begin{cases} \sqrt{\mu_A(x)/2} & \text{if } \mu_A(x) \leq 0.5 \\ 1 - \sqrt{(1-\mu_A(x))/2} & \text{otherwise} \end{cases} \tag{2.54}$$

where $p > 1$ may be tuned to control the intensity of the hedges in Equations (2.51) through (2.53). Concentration hedges, corresponding to linguistic terms such as *very*, *greatly* and *decidedly*, create modified membership functions where boundaries are shifted in favour of higher membership values. Dilation hedges have the opposite effect and correspond to terms such as *somewhat*, *sort of* and *fairly*. Terms such as *indeed* and, for higher values of p, *extremely*, correspond to intensification hedges which emphasise contrast. Finally, blurring hedges, corresponding to terms such as *seldom* and *more or less*, perform the opposite of intensification by introducing vagueness.

2.5.2 Fuzzy Controllers

Figure 2.14 outlines a simple architecture for a fuzzy controller [75] consisting of three primary components. First, the condition interface, which is responsible for converting outputs from the system into a fuzzy form, hence the term fuzzifier, utilised by the fuzzy inferencing engine. Next, the engine performs inferencing, based on linguistic rules, to determine an appropriate control action. Finally, the action interface is responsible for interpreting the output of the inferencing process and converting it back into system specific actions through a process known as defuzzification. Thus, a feedback loop is realised where the controller constantly monitors the system while effecting control actions on the system according to its rule base.

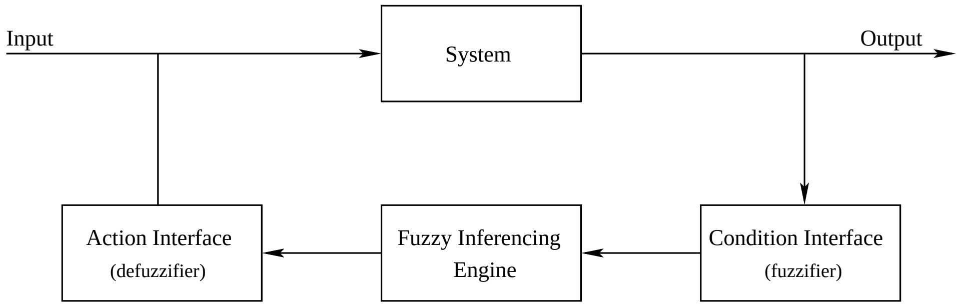

Figure 2.14: Fuzzy Controller Architecture

As a somewhat contrived example, consider a fuzzy system used to control a hypothetical cigarette dispensing machine. Rather than blindly supplying smokers with their selection, this particular machine is designed to wean them off their addiction by carefully limiting their supply of cigarettes. Further, assume that a domain expert, such as a lung specialist, has provided a number of linguistic rules. For example, “If the user is very old and a regular smoker then dispense as many cigarettes as requested.” The reasoning behind such rule might be that a heavy smoker who has managed to survive to a ripe old age is likely to die of natural causes long before contracting lung cancer. Other rules might curtail the number of cigarettes dispensed to younger smokers depending on their average intake, or limit the provision to zero for casual smokers.

The dispensing machine provides the controller with two inputs requiring fuzzification, the actual age of the user and the average number of cigarettes consumed on a daily basis. Fuzzification entails identifying the fuzzy sets used by the inferencing engine and calculating the degrees of membership to each of these sets given the inputs. Continuing with the example rule, according to Figure 2.13, the membership function for the set corresponding to the linguistic term *very old* is given by:

$$\begin{aligned}\mu_{[\text{very old}]}(x) &= \mu_{\text{old}}(x)^2 \\ &= \begin{cases} \left(e^{-(x-100)^2/1000}\right)^2 & \text{if } x \leq 100 \\ 1 & \text{otherwise} \end{cases}\end{aligned} \tag{2.55}$$

where x is the actual age of the user and the concentration hedge for the term *very* is assumed to be implemented with $p = 2$. A membership function for $\mu_{[\text{regular smoker}]}$ can be defined in a similar fashion.

After fuzzifying the inputs, the next step is to perform inferencing using the fuzzy rule base. Typically, the rule base is made up of a list of rules of the form:

$$\text{if } \textit{antecedent} \longrightarrow \textit{consequent} \tag{2.56}$$

where the *antecedent* consists of one or more fuzzy sets combined using the operators in Table 2.1 to form a logical expression. In the case of a Mamdani [75] controller, the *consequent* consists of a single target fuzzy set. The value of the antecedent, also known as the firing strength of the rule, determines the degree of membership to the target set in the consequent. A Takagi-Sugeno [110] controller, on the other hand, permits higher order consequents.

The antecedent for the example sentence presented earlier may be calculated as either:

$$\mu_{[\text{very old}]}(x) \cap \mu_{[\text{regular smoker}]}(y) = \min\{\mu_{[\text{very old}]}(x), \mu_{[\text{regular smoker}]}(y)\} \tag{2.57}$$

or, the product:

$$\mu_{[\text{very old}]}(x) \cap \mu_{[\text{regular smoker}]}(y) = \mu_{[\text{very old}]}(x)\mu_{[\text{regular smoker}]}(y) \tag{2.58}$$

depending on the choice of intersection operator, where x and y are the age and average daily cigarette consumption respectively. The firing strengths for the remaining antecedents in the rule base are calculated in a similar fashion.

The defuzzification processes is performed for each output linguistic variable to determine a single non-fuzzy, or crisp, value to feed back to the system. In the example rule, the linguistic variable associated with the cigarette limit has a consequent of *unlimited*, however, this must still be combined in a sensible way with the consequents of any other rules pertaining to the same linguistic variable.

Various defuzzification strategies may be employed, the height of the centroid under the composite area defined by the chosen strategy is used as the crisp action result:

- **max-min strategy:** Only the membership function of the consequent associated with the rule having the highest firing strength is used.
- **averaging strategy:** All membership functions pertaining to the linguistic variable in question are clipped at the average firing strength of the combined rules.
- **root-sum-square strategy:** All membership functions pertaining to the linguistic variable in question are scaled to the firing strengths of their respective rules.
- **clipped centre of gravity:** All membership functions pertaining to the linguistic variable in question are clipped at the firing strengths of their respective rules.

Thus, all the consequents corresponding to a given linguistic variable are combined, based on the chosen defuzzification strategy, into a single crisp value. At the one extreme, the max-min strategy only takes into account the most dominant rule, while the averaging strategy dilutes the result, giving no preference to rules with higher firing strength. Further, it is possible to bias the rules, by scaling their firing strengths, based on the confidence placed on a given rule by a human expert.

2.6 Other Paradigms

One specific example of a relatively new CI paradigm is the Artificial Immune System (AIS) [24], which is a computational pattern recognition technique, based on how white blood cells in the human immune system detect pathogens that do not belong to the body. Instead of building an explicit model of the available training data, an AIS builds an implicit classifier that models everything else but the training data, making it suited to detecting anomalous behaviour in systems. Thus, an AIS is well suited for applications in anti-virus software, intrusion detection systems and fraud detection in the financial sector.

Further, fields such as Artificial Life (ALife), robotics (especially multi-agent systems) and bioinformatics are application areas for CI techniques. Alternatively, it can be argued that those fields are a breeding ground for tomorrow's CI ideas.

For example, evolutionary computing techniques have been successfully employed in bioinformatics to decipher genetic sequences [35]. Hand in hand with that comes a deeper understanding of the biological evolutionary process and improved evolutionary algorithms.

As another example, consider RoboCup[1], a project with a very ambitious goal. The challenge is to produce a team of autonomous humanoid robots that will be able to beat the human world championship team in soccer by the year 2050. This is obviously an immense undertaking that will require drawing on many disciplines. The mechanical engineering aspects are only one of the challenges standing in the way of meeting this goal. Controlling the robots is quite another. Swarm robotics [6, 99], an extension of swarm intelligence into robotics, is a new paradigm in CI that may hold some of the answers. In the mean time, simulated RoboCup challenges, which are held annually, will have to suffice.

2.7 Hybrid Approaches

Attempting to produce an exhaustive list of all the possible hybrid approaches here is certainly an exercise in futility. There are, simply stated, so many ways in which different CI techniques can be combined that any attempt to survey them would probably require an entire dissertation dedicated to that task alone. Indeed, hybrid approaches need not even limit themselves to combining techniques drawn from the CI discipline alone, making the possibilities virtually endless. Instead, the purpose of this section is to emphasise the existence of hybrids, by means of a few examples, and to highlight the importance of a flexible software framework which enables composing various techniques together in new and interesting ways.

As a first example, consider the PSO, discussed in Section 2.4.1. One hybridised approach, dubbed the Dissipative PSO (DPSO) [122], builds on concepts borrowed from thermodynamics. The designers of the DPSO noted that the self organising nature of the PSO, where particles follow an irreversible process towards higher fitness, ultimately

[1]http://www.robocup.org

lacks the capability for sustainable development. By introducing negative entropy into the algorithm and operating as a dissipative structure, the DPSO is able to maintain swarm diversity and improve the quality of solutions found by the search. Now, while it could be argued that the DPSO is not a true hybrid but rather a relatively simple extension of the PSO, the relevant issue is that a software implementation should, as far as possible, reuse an existing implementation of the PSO and simply compose it with something that implements the dissipative capability.

Another method to hybridise the PSO is to update the positions of the best performing particles using a different optimisation process. Consider the velocity update in Equation (2.41), the best particles in their respective neighbourhoods will have $\mathbf{x} = \mathbf{y} = \hat{\mathbf{y}}$, resulting in zero cognitive and social components. Eventually, the velocity components will also degrade to zero, since $0 \leq w < 1$, and these particles will stop moving. Further, it is possible for the rest of the particles to collapse onto these positions too, resulting in stagnation of the entire swarm. The Guaranteed Convergence PSO (GCPSO) [114, 113] replaces the velocity update for the neighbourhood best particles with a modified unimodal optimiser [103], in effect creating a hybrid of the two. Properties of the GCPSO include rapid convergence and a guarantee to at least converge onto a locally optimum solution. Once again, a software implementation should make provision for this kind of hybrid, perhaps by having a pluggable optimisation process for the neighbourhood best, or indeed any particle. This kind of flexibility would enable the optimisation process for any particle to be replaced by say, gradient descent, LeapFrog [102], an evolutionary algorithm, or perhaps even another PSO to create a hierarchical PSO-PSO hybrid. Further, it may be desirable to simultaneously compose GCPSO and DPSO into yet another hybrid.

As hinted in Sections 2.1.3 and 2.2.1, neural networks present another opportunity for hybridisation. By representing network weights as a single vector and the SSE over the training set as an objective function, neural network training can be re-framed as an optimisation problem. This opens the door for many hybrids, including using GAs, EP, ES, cultural evolution or PSOs to train neural networks. Again, a software implementation should enable neural network training using any optimisation algorithm in a flexible fashion.

One specific hybrid example, which spans multiple paradigms, is Blondie 24 [34]. Blondie 24 is an advanced game playing framework with the ability to understand and develop strategies for a game given only its rules as prior knowledge. The framework draws on three paradigms: game theory, neural networks and evolutionary computing. The approach involves evaluating a traditional game tree [85] using a neural network as an evaluation function. In order to find the optimal network for the task, Blondie 24 employs a competitive coevolutionary approach to evaluate network against network. Over time, neural networks evolve that are better able to evaluate the game state and as a result become stronger players. This approach has been taken one step further [79, 40] by extending the coevolutionary approach to particle swarms, producing a four way game tree, neural network, coevolution, PSO hybrid. Designing software flexible enough to support such hybrids is a challenging task.

Other hybrid approaches include fuzzy neural networks [88, 129], a breeding PSO that leverages evolutionary crossover [74] and evolutionary processes for learning rules for fuzzy controllers [22].

2.8 Software Requirements

Section 2.7 illustrated the importance of a flexible software framework. It should be possible to reuse and compose various algorithms in different ways with a minimum amount of recoding. Ideally, any permutation should be made possible by merely changing the configuration of the system at runtime.

Section 2.1 demonstrated that most problem classes can be re-framed as optimisation problems. For this reason, any optimisation algorithm should be able to operate on any problem which can be cast as an optimisation problem, as defined in Section 2.1.1.

It is tempting to make the next step and simply treat all problems as optimisation problems, that way the interface between algorithms and problems is reduced to a single set of interactions. To see why this is a poor idea, consider what the interface for an optimisation problem might look like. Optimisation algorithms, such as the PSO or a GA, require only two pieces of information from the problem. Firstly, they need to know the domain of the problem. Secondly, and most importantly, they need to know the fitness of a potential solution to the problem. Any more information would not be used by such optimisation algorithms. Indeed, many optimisation problems, such as function minimisation, simply cannot provide any more information either. Thus, an optimisation problem must be characterised by an interface that supplies the domain of the problem and the fitness of a given solution within that domain.

From an implementation perspective, contrast the functioning of a generic optimisation algorithm, which only needs to query the fitness of potential solutions to a problem, with a feed forward neural network. The neural network needs access to a set of training patterns with their associated inputs and targets. Thus, the neural network requires more information from the problem domain than a generic optimisation algorithm, which is satisfied with only having access to an objective function. Therefore, the software should have different interfaces for problems that make different information available to algorithms according to their context. The various algorithms, in turn, should be able to be applied to whatever types of problems they support, also by means of configuration at runtime. Further, any problem that can be represented as another type of problem, via some transformation such as those discussed in Section 2.1, should expose an interface to do so. For example, a TSP should expose an optimisation problem interface in addition to its more natural interface, which would expose a graph topology necessary for an algorithm such as ACO.

Stopping conditions are another important element of algorithms that should be handled in a pluggable way. All algorithms presented in this chapter loop until some stopping condition is met. Those stopping criteria exist independently of the particular algorithm. Any algorithm can have as a stopping criterion a maximum number of iterations. Optimisation algorithms may have as a stopping criterion a maximum number of evaluations of the objective function. Particle swarms may have a stopping criterion based on a minimum swarm diameter. Once again, stopping criteria should be configurable at runtime for any algorithm.

Finally, since the software will be used for scientific research it is important to be able to measure certain properties during the execution of any algorithm. Some of these properties may be dependent on the specific problem or algorithm being used, however, they should still be implemented in a reusable fashion externally to the algorithm. Measurements should not clutter the implementation of algorithms and should not even be present if they are not used, for example, if the software is deployed in a specific non-research application that has no need for measurements.

Creating a flexible software design is a challenging task. The next chapter presents patterns which are invaluable aids for creating such designs.