9 Particle swarm optimization

This chapter covers

- Introducing swarm intelligence
- Understanding the continuous particle swarm optimization algorithm
- Understanding binary particle swarm optimization
- Understanding permutation-based particle swarm optimization
- Adapting particle swarm optimization for a better trade-off between exploration and exploitation
- Solving continuous and discrete problems using particle swarm optimization

In the treasure-hunting mission I introduced in chapter 2, suppose you want to collaborate and share information with your friends instead of doing the treasure-hunting alone. However, you do not want to follow a competitive approach in which you only keep better-performing hunters and recruit new hunters to replace poorer-performing ones, like in the genetic algorithm (GA) explained in the previous chapters. You want to adopt a more cooperative approach and keep all the hunters, without replacing any, but you want to give more weight to the better-performing

hunters and try to emulate their success. This scenario uses *swarm intelligence* and corresponds to population-based optimization algorithms such as *particle swarm optimization* (PSO), *ant colony optimization* (ACO), and *artificial bee colony* (ABC) algorithms, which will be explained in this fourth part of the book.

In this chapter, we'll focus on different variants of PSO algorithms and apply them to solve continuous and discrete optimization problems. These variants include continuous PSO, binary PSO, permutation-based PSO, and adaptive PSO. Function optimization, the traveling salesman problem, neural network training, trilateration, coffee shop planning, and the doctor scheduling problem are discussed in this chapter and its supplementary exercises included in appendix C. The next chapter will cover the ACO and ABC algorithms.

9.1 Introducing swarm intelligence

Life on this planet is full of astonishing examples of collective behavior. Individual species depend upon one another for sustenance, often forming surprising alliances to achieve a common goal: the continuance of the species. The majority of living things also display amazing altruism in order to protect and provide the best care for their offspring, comparable to any form of sacrifice shown by human beings. They can cooperatively perform complex tasks such as foraging for food, dividing up labor, constructing nests, brood sorting, protecting, herding, schooling, and flocking, to name just a few. These complex collective behaviors emerge from individual interactions between spatially distributed and simple entities without a centralized controller or coordinator and without a script or access to global information. Various cooperation patterns, communication mechanisms, and adaptation strategies are employed to enable such complex collective behaviors.

Swarm intelligence

Swarm intelligence is a subfield of artificial intelligence that explores how large groups of relatively simple and spatially distributed agents can interact with each other and with their environment in a decentralized and self-organized manner to collectively achieve complex goals.

Several efficient population-based algorithms have been designed to exploit the power of collective intelligence by mimicking the collective behaviors observed in nature to solve complex optimization problems. Table 9.1 provides a non-exhaustive list of swarm intelligence algorithms and their sources of inspiration from unicellular and multicellular living organisms.

Table 9.1 Examples of swarm intelligence algorithms and their sources of inspiration

Organisms	Class	Source of inspiration	Algorithms
Unicellular organisms	Bacteria	Bacterial swarm foraging	Bacterial foraging optimization algorithm (BFO)
			Bacterial swarming algorithm (BSA)
Multicellular organisms	Bird/fish	Bird flocking and fish schooling	Particle swarm optimization (PSO)
	Ant	Ant foraging behaviors	Ant colony optimization (ACO)
	Bees	Foraging behavior of honeybees	Artificial bee colony (ABC)
	Bats	Echolocation behavior of bats	Bat algorithm (BA)
	Fireflies	Flashing behavior of fireflies	Firefly algorithm (FA)
	Butterflies	Foraging behavior of butterflies	Butterfly optimization algorithm (BOA)
	Dragonflies	Static and dynamic swarming behaviors of dragonflies	Dragonfly algorithm (DA)
	Spiders	Cooperative behavior of the social spiders	Social spider optimization (SSO)
	Krill	Herding behavior of krill	Krill herd (KH)
	Frogs	Frog cooperative search for food	Shuffled frog leaping algorithm (SFLA)
	Fish	Gregarious behavior of fish	Fish school search (FSS)
	Dolphins	Dolphins' behavior in detecting, chasing after, and preying on swarms of sardines	Dolphin partner optimization (DPO)
			Dolphin swarm optimization algorithm (DSOA)
	Cats	Resting and tracing behaviors of cats	Cat swarm optimization (CSO)
	Monkeys	Search for food	Monkey search algorithm (MSA)
	Lions	Solitary and cooperative behaviors of lions	Lion optimization algorithm (LOA)
	Cuckoos	Reproduction strategy of cuckoos	Cuckoo search (CS)
			Cuckoo optimization algorithm (COA)
	Wolves	Leadership hierarchy and hunting mechanism of gray wolves	Wolf search algorithm (WSA)
			Grey wolf optimizer (GWO)

For example, bacteria, which are single-celled organisms, possess underlying social intelligence allowing them to cooperate to solve challenges. Bacteria develop intricate communication capabilities like chemotactic signaling to cooperatively self-organize into highly structured colonies with elevated environmental adaptability. Bacterial chemotaxis is the process by which bacterial cells migrate through concentration gradients of chemical attractants and repellents. The E. coli bacterium uses this bacterial chemotaxis during the foraging process. This collective behavior provides the basis for optimization algorithms such as the bacterial foraging optimization algorithm (BFO) and bacterial swarming algorithm (BSA).

Ethology, the study of animal behavior, is the main source of inspiration for swarm intelligence algorithms such as particle swarm optimization (PSO), ant colony optimization (ACO), artificial bee colony (ABC), bat algorithm (BA), firefly algorithm (FA), and social spider optimization (SSO). For example, honeybees are highly cooperative social insects that cooperatively construct hives in which about 30,000 bees can live and work together. They differentiate their work: some make wax, some make honey, some make bee-bread, some shape and mold combs, and some bring water to the cells and mingle it with the honey. Young bees engage in out-of-door work while the elder bees do the indoor work. During the foraging process, rather than expending energy searching in all directions, honeybee colonies use individual foragers to reduce the cost/benefit ratio. Additionally, colonies concentrate their foraging efforts on the most lucrative patches and disregard those of lesser quality. It has been observed that when a colony's food resources are scarce, foragers recruit more nestmates to food sources they have found, and changes in their dance patterns upon returning to the hive facilitate this increased recruitment.

The fundamental components of swarm intelligence algorithms typically involve numerous decentralized processing agents that operate without central supervision. These agents communicate with neighboring agents and adapt their behavior based on received information. Furthermore, the majority of the research carried out on swarm intelligence algorithms is primarily based on experimental observations of collective behavior exhibited by living organisms. These observations are translated into models, which are then tested through simulations in order to derive the metaheuristics that form the basis of swarm intelligence algorithms, as illustrated in figure 9.1. This experimental approach enables researchers to gain a deeper understanding of the complex interactions between individual agents and how they give rise to collective behavior. By simulating these interactions and testing various scenarios, researchers can refine the algorithms and improve their effectiveness. As an example of such experimental research, you can watch the "The Waggle Dance of the Honeybee" video of the experiment conducted by Georgia Tech College of Computing to understand how honeybees communicate the location of new food sources (www.youtube.com/watch?v=bFDGPgXtK-U).

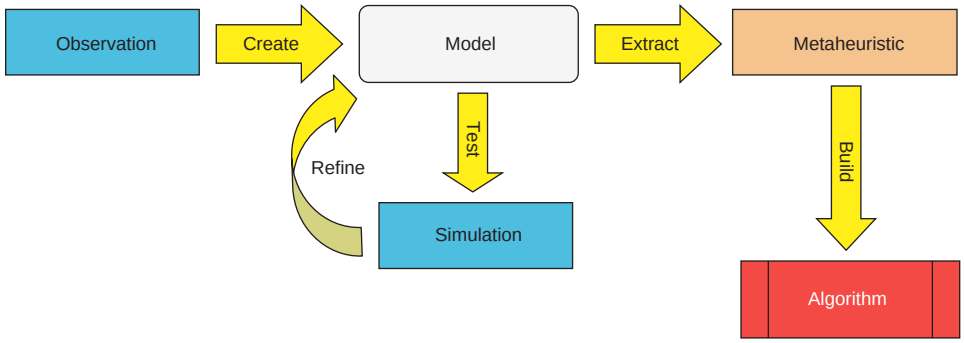


Figure 9.1 Derivation process for swarm intelligence algorithms

Algorithm 9.1 shows the common steps in a swarm intelligence algorithm. The algorithm starts by initializing the algorithm parameters, such as the number of individuals in the swarm, the maximum number of iterations, and the termination criteria. A swarm of initial candidate solutions is then sampled (the different sampling methods were explained in section 7.1). The algorithm then iterates over all the individuals in the swarm, performing the following operations: finding the best so far, finding the best neighbor, and updating the individual.

Algorithm 9.1 Swarm intelligence algorithm

```
Initialize parameters
Initialize swarm
While (stopping criteria not met) loop over all individuals
    Find best so far
    Find best neighbor
    Update individual
```

The individual and the neighbor are evaluated using the defined objective/fitness function. The neighborhood structure and update mechanism depend on the algorithm being used. This loop over all the individuals is repeated until the termination criterion is met, which could be a maximum number of iterations or reaching a satisfactory fitness level. At this point, the algorithm stops and returns the best solution found during the optimization process. In the following sections, we'll dive deep into the PSO algorithm.

9.2 Continuous PSO

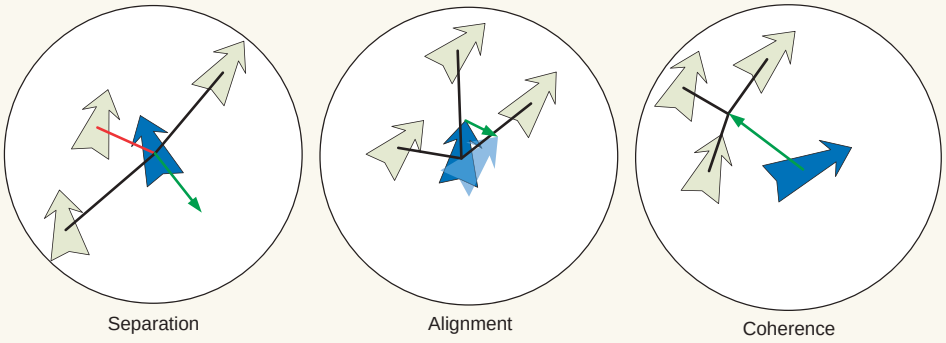
Particle swarm optimization (PSO) is a population-based stochastic optimization technique developed by Russell Eberhart and James Kennedy in 1995. Since then, PSO has gained popularity and has been applied to various real-world applications in different domains. This algorithm is inspired by the conduct of social organisms such as birds, fish, ants, termites, wasps, and bees. PSO emulates the actions of these creatures, with

each member of the swarm being referred to as a *particle*, akin to a bird in a flock, a fish in a school, or a bee in a colony. Eberhart and Kennedy opted to use the term "particle" to refer to an individual or candidate solution in the context of optimization, as they believed it was more suitable for describing the particle's velocity and acceleration.

Bird flocking

Bird flocking is a behavior controlled by three simple rules, as illustrated in the following figure:

- *Separation*—Prevent getting too close to nearby birds to avoid overcrowding.
- *Alignment*—Adjust the heading to correspond to the average direction of neighboring birds.
- *Coherence*—Move toward the average position of neighboring birds.



Bird flocking rules: separation, alignment, and coherence

When birds—spatially distributed agents interacting with each other and with their environment in a decentralized and self-organized manner without access to global information—apply these three simple rules, the outcome is the emergent behavior of bird flocking.

The particles (candidate solutions) move, or fly, through the feasible search space by following the current best particles. Thus, PSO is guided by a straightforward principle: emulate the success of neighboring individuals. Each particle in the swarm operates in a decentralized manner by utilizing both its own intelligence and the collective intelligence of the group. Therefore, if one particle uncovers a favorable route to food, the remaining members of the swarm can immediately adopt the same path.

The PSO algorithm

"This [PSO] algorithm belongs ideologically to that philosophical school that allows wisdom to emerge rather than trying to impose it, that emulates nature rather than trying to control it, and that seeks to make things simpler rather than more complex." J. Kennedy and R. Eberhart, inventors of PSO [1].

Figure 9.2 shows the PSO flowchart. We start by initializing the algorithm parameters and creating an initial swarm of particles. These particles represent the candidate solutions. Each particle in the search space holds the current position x^i and current velocity v^i. The fitness of each particle is then evaluated based on the fitness/objective function to be optimized. The best position each particle has achieved so far is known as the personal best or *pbest*. The best position achieved by the particles in its neighborhood is known as *nbest*. If the neighborhood is restricted to a few particles, the best is called the local best, *lbest*. If the neighborhood is the whole swarm, the best achieved by the whole swarm is called the global best, *gbest*. We'll discuss neighborhood structures in PSO further in section 9.2.3.

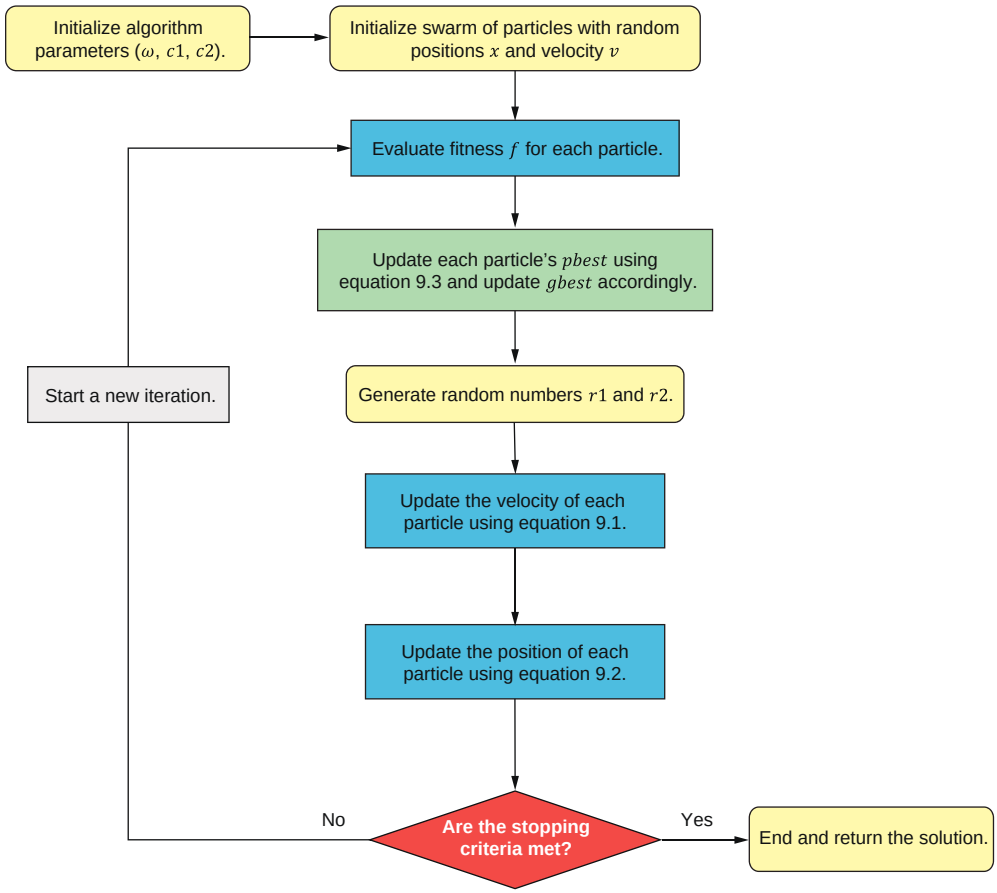


Figure 9.2 The PSO algorithm

After evaluating the fitness of each particle, PSO updates each particle's personal best position if the current fitness is superior, identifies the global best position based on the best fitness in the entire swarm, and adjusts particle velocities and positions using a

combination of personal and global information. These steps guide the swarm toward optimal or near-optimal solutions by balancing individual and collective learning, promoting exploration and exploitation in the search space. The process iterates until termination criteria are met.

9.2.1 Motion equations

The velocity (v) and position (x) of each particle are updated using the following equations:

$$v_{k+1}^{id} = \omega \times v_k^{id} + c1r1_k^d \left(pbest_k^{id} - x_k^{id}\right) + c2r2_k^d \left(gbest_k^d - x_k^{id}\right) \quad \textbf{9.1}$$

$$x_{k+1}^{id} = x_k^{id} + v_{k+1}^{id} \quad \textbf{9.2}$$

where

- k is the iteration number.
- i and d are the particle number and the dimension. For example, dimension = 1 in the case of a univariate optimization problem with a single decision variable, dimension = 2 in the case of a bivariate problem, etc.
- ω is the inertia weight.
- $c1$, $c2$ are the acceleration coefficients.
- $r1$, $r2$ are random numbers between 0 and 1 and are generated in each iteration for each dimension, and not for each particle.
- *pbest* is the best position achieved by the particle.
- *gbest* is the best position achieved by the whole swarm. *gbest* should be replaced by *nbest or lbest* if you are dividing the swarm into multiple neighborhoods.

As you can see in these two equations, we start by updating the velocity $v_{(k+1)}{}^{id}$. The position is then updated to $x_{(k+1)}{}^{id}$ by taking the current position x_k^{id} and adding to it the new displacement $v_{(k+1)}{}^{id} \times timestamp$ where $timestamp = 1$, which represents a single iteration.

To understand these motion update equations, let's visualize these equations using vectors in the 2D Cartesian coordinate system, as shown in figure 9.3. As you can see, the velocity update equation consists of three primary components, each contributing to the movement of particles in the search space:

- *Inertia component*—The first part of the velocity update equation represents the influence of a particle's inertia, taking into account that a particle (like a fish in a school or a bird in a flock) cannot abruptly change its direction. As you will see later, this inertia component is crucial, as it allows the algorithm to be more adaptive and helps maintain a balance between exploration and exploitation.

- *Cognitive component*—The second part of the equation, referred to as the cognitive component, represents the particle's attraction toward its personal best position, or individual proximity (*i*-proximity). This component reflects the degree of trust a particle places in its own past experiences, without considering the experiences of its neighbors or the swarm as a whole. The cognitive component encourages particles to explore the areas around their personal best positions, allowing them to fine-tune their search in promising regions.
- *Social component*—The third part of the velocity update equation is the social component, which represents the particle's attraction to the swarm's collective knowledge or group proximity (*g*-proximity). This component takes into account the experiences of neighboring particles and the swarm as a whole, guiding the particles toward the global best position found so far. The social component fosters collaboration among particles, helping them converge toward an optimal solution more effectively.

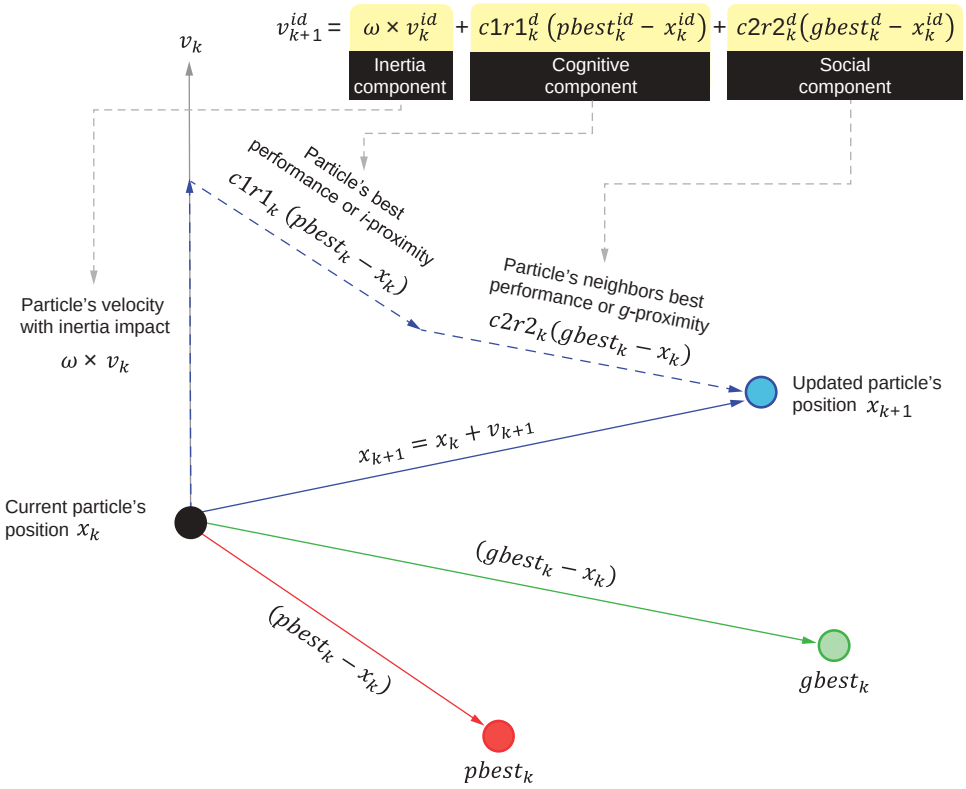


Figure 9.3 Visualizing the motion equation for a particle in the swarm

To better understand the meaning of each component, imagine a group of friends visiting a large amusement park for the first time. Their goal is to visit the most thrilling rides in the park as efficiently as possible. The friends can be thought of as particles in

the PSO algorithm, with each person's enjoyment of the rides serving as the objective function to optimize. Each person has a preferred way of exploring the available rides, like walking through certain parts of the park or trying specific rides like roller coasters or water slides. This is similar to the inertia component in PSO, where particles maintain their current velocity and direction, ensuring they don't change their exploration pattern too abruptly.

Each friend relies on their own personal experiences to find the most thrilling rides. For instance, one friend might have had a great time on a roller coaster earlier in the day. They're more likely to return to their favorite one or want to find more similar rides, knowing it was a good choice. They trust their judgment and focus on exploring the areas around the roller coaster, seeking out rides that they think they'll enjoy based on their personal experience. This is the cognitive component, where particles in PSO are attracted to their personal best positions, following their past experiences and individual preferences.

The friends then collaborate to find the most thrilling ride based on their shared experiences. Imagine one of the friends has just ridden the most exciting roller coaster and can't wait to tell the others about it. As they share their excitement, the group collectively becomes more attracted to that ride, influencing their individual choices. This is the social component, where particles in PSO are influenced by the global best position or the collective knowledge of the swarm.

The following subsections dive into more detail about the different PSO parameters.

9.2.2 Fitness update

After moving, each particle updates its own personal best using the following equation, assuming a minimization problem:

$$pbest_{k+1}^{id} = \begin{cases} x_{k+1}^{id} \; if \; f(x_{k+1}^{id}) \leq f(pbest_k^{id}) \\ pbest_k^{id} \qquad otherwise \end{cases} \tag{9.3}$$

After that, each neighborhood updates its best as follows:

$$nbest_{k+1}^{id} = \underset{nbest_{k+1}^{id} \in N}{\arg\min} \; f\left(pbest_{k+1}^{id}\right) \tag{9.4}$$

The neighborhood best (*nbest*) is the same as the global best (*gbest*) if the neighborhood is the whole swarm.

PSO has two main variants based on how particles' positions and velocities are updated—synchronous and asynchronous PSO:

- *Synchronous PSO (S-PSO)*—All particles in the swarm update their positions and velocities simultaneously in a global manner. The local and global best are then updated. This synchronous approach ensures that all particles have access to the same global best position when updating their velocities and positions, promoting global exploration.

- *Asynchronous PSO (A-PSO)*—Particles are updated based on the current state of the swarm. This asynchronous approach allows the particles to update their positions and velocities based on the most recent information available.

Figure 9.4 shows the difference between S-PSO and A-PSO. As you'll notice, in A-PSO, the neighborhood best update is moved into the particle's update loop. This allows particles to evaluate their fitness and update their positions and velocities independently and asynchronously.

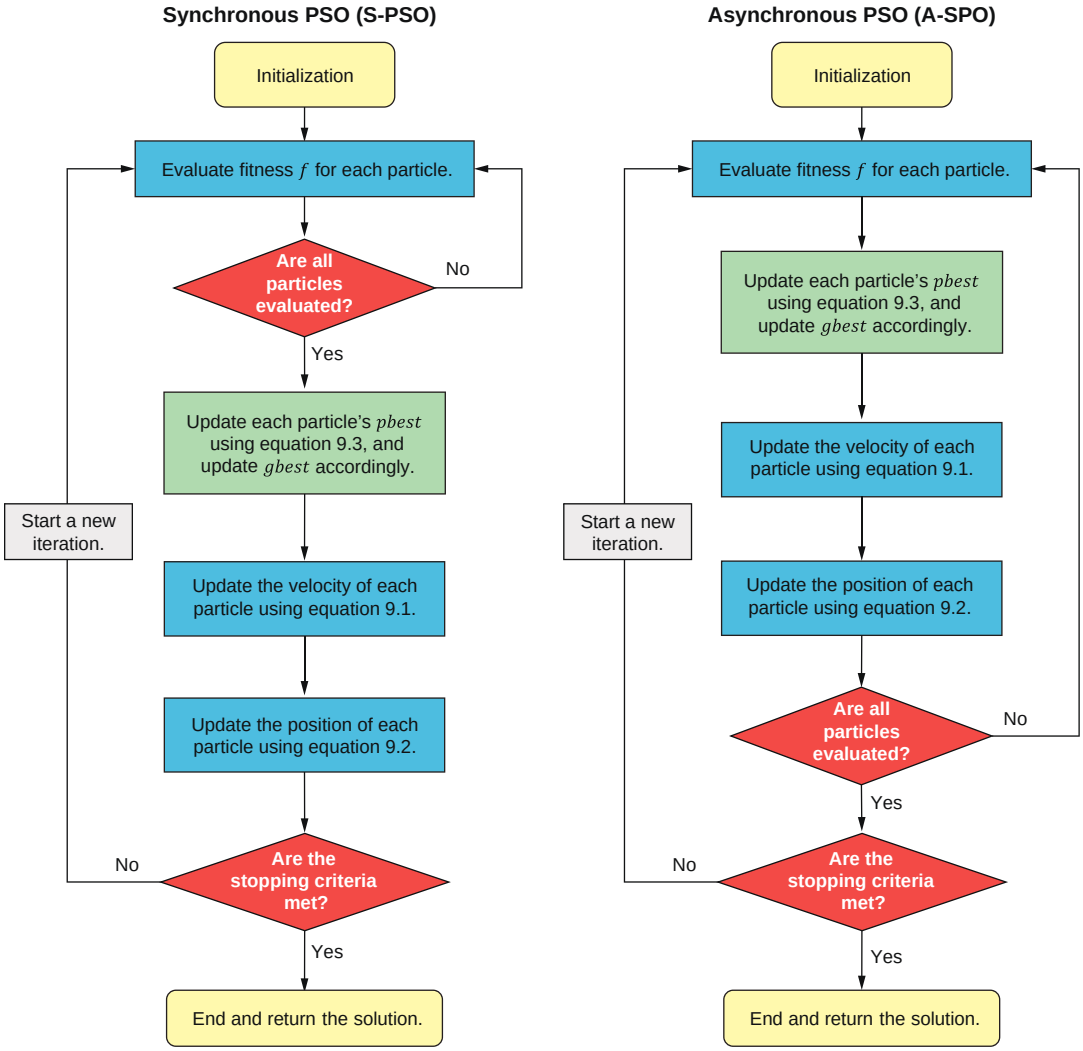


Figure 9.4 Synchronous and asynchronous PSO

Although both synchronous and asynchronous PSO strategies can be employed to handle optimization problems, the asynchronous version is generally more effective, as it allows particles to take advantage of the most recent neighbor information.

9.2.3 Initialization

PSO initialization includes initializing the particle position, velocity, and personal best, and initializing the algorithm's parameters:

- *Particle position initialization*—Particle positions represent candidate solutions to the problem, and they can be sampled using different sampling methods, as explained in section 7.1. For example, the initial positions of the particles can be randomly assigned within the defined feasible search space.
- *Particle velocity initialization*—The velocities of the particles can be set to zero or small values initially. Initializing them with small velocities ensures that the particles' updates are gradual, preventing them from moving too far away from their starting positions. In contrast, large initial velocities may lead to significant updates, which can potentially cause divergence and hinder the convergence of the algorithm.
- *Personal best position initialization*—The personal best position of each particle, which represents the best solution found by the particle so far, should be initialized to its initial position. This allows the particles to begin their search with their starting points as a reference and to update their personal bests as they discover better solutions.

As shown in equation 9.1, PSO has three primary parameters that play a critical role in controlling the search algorithm's behavior: the inertia weight (ω) and the acceleration coefficients ($c1$, $c2$). These parameters influence the balance between exploration and exploitation within the optimization process:

- *Inertia weight*—Large values of ω encourage exploration, while small values promote exploitation, allowing the cognitive and social components to exert greater control. A widely adopted value for ω is 0.792.
- *Acceleration coefficients*—Setting $c1$ to 0 reduces the PSO algorithm to a *social-only* or *selfless PSO* model. In this case, particles are solely attracted to the group best and ignore their personal bests. This leads to an emphasis on global exploration based on the swarm's collective knowledge. Setting $c2$ to 0 results in a *cognition-only model*, where particles act as independent hill climbers, relying only on their personal bests. In this scenario, the particles do not consider the experiences of other swarm members, focusing on local exploitation based on their individual experiences. In many applications, $c1$ and $c2$ are set to 1.49. Although there is no theoretical justification for this specific value, it has been empirically found to work well in various optimization problems. Generally, the sum of $c1$ and $c2$ should be less than or equal to 4 to maintain the algorithm's stability and convergence properties.

Other parameters to consider include swarm size and neighborhood size. There is no one-size-fits-all solution, as the optimal values depend on the specific problem being solved. However, some best practices and guidelines can help inform your choices:

- *Swarm size*—A large swarm size can promote global exploration and prevent premature convergence, but at the cost of increased computational effort. A small swarm size can lead to faster convergence and reduced computational effort but may increase the risk of premature convergence. For many problems, a swarm size between 20 and 100 particles has been found to yield good results. It is advisable to conduct experiments with different swarm sizes to determine the best trade-off between exploration, exploitation, and computational complexity for the problem at hand.
- *Neighborhood size*—A large neighborhood size can encourage global exploration and information sharing among particles but may reduce the ability to exploit local optima. A small neighborhood size can promote local exploitation and convergence speed but may limit global exploration. You can use different neighborhood structures, as you'll see in the next subsection.

Generally speaking, selecting the best algorithm parameters requires experimentation and fine-tuning based on the specific problem you are trying to solve. It is often beneficial to perform a sensitivity analysis or use a parameter-tuning technique to find the optimal parameter values for your problem. We'll look at this in more detail in section 9.5.

9.2.4 Neighborhoods

In the PSO algorithm, particles within a specific vicinity engage in mutual communication by sharing details about each other's success within that local region. Subsequently, all particles gravitate toward a position that is considered to be an improvement based on a key performance indicator. The efficacy of the PSO algorithm is heavily reliant on the social network structure it employs. Choosing an appropriate neighborhood topology plays a crucial role in ensuring the algorithm's convergence and in preventing it from becoming trapped in local minima.

Some of the prevalent neighborhood topologies utilized in PSO include the star social structure, the ring topology, the Von Neumann model, and the wheel topology:

- *The star social structure, also known as the global best (gbest) PSO*—This is a neighborhood topology where all particles are connected, as shown in figure 9.5a. This structure allows access to global information within the swarm, with the result that each particle is drawn toward the optimal solution discovered by the entire swarm. The *gbest* PSO has been demonstrated to converge more rapidly than other network structures. However, it is more prone to becoming ensnared in local minima without fully exploring the search space. This topology excels when applied to unimodal problems, as it allows for efficient and effective optimization in such cases.

- *Ring topology, also known as the local best (lbest) PSO*—Following this topology, a particle interacts exclusively with its immediately adjacent neighbors (figure 9.5b). Each particle endeavors to emulate its most successful neighbor by gravitating toward the optimal solution discovered within the local vicinity. Although convergence occurs at a slower pace than with the star structure, the ring topology explores a more extensive portion of the search space. This topology is recommended for multimodal problems.
- *Von Neumann model*—In this topology, particles are arranged in a grid-like structure or square topology where each particle is connected with four other particles (the neighbors above, below, and to the right and left), as shown in figure 9.5c.
- *Wheel topology*—In this topology, the particles are isolated from each other, and one particle is randomly selected as the focal point or hub for all information flow, as illustrated in figure 9.5d.

The choice of neighborhood topology depends on the problem's characteristics and the desired balance between exploration and exploitation. Experiment with different topologies to find the best fit for your problem.

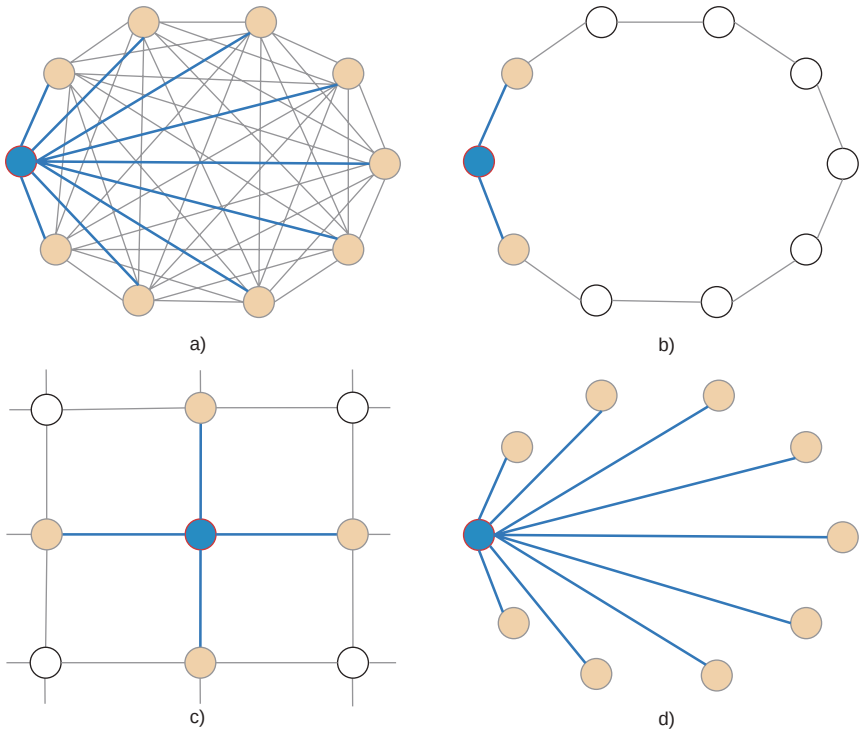


Figure 9.5 PSO neighborhood topologies: a) the star social structure, b) the ring topology, c) the Von Neumann model, and d) the wheel topology

Let's now look at how to solve a continuous optimization problem using PSO. The Michalewicz function is a nonconvex mathematical function commonly used as a test problem for optimization algorithms. This function is given by the following formula:

$$f(x) = -\sum_{j=1}^{d} \sin(x_j)\left[\sin\left(\frac{jx_j^2}{\pi}\right)\right]^{2m} \quad \textbf{9.5}$$

where d is the dimension of the problem and m is a constant (usually $m = 10$). This function has d local minima. For $d = 2$, the minimum value is –1.8013 at (2.20, 1.57).

Let's start by defining the Michalewicz function as shown in listing 9.1. This function can accept size-1 arrays with single or multiple rows. If the input position is a size-1 array with a single row, we reshape it to a 2D array with a single row using the `reshape()` function. A *size-1 array*, also known as a *singleton array*, is an array data structure that contains only one element. This reshaping enables uniform handling of both single-row size-1 arrays and arrays with multiple rows. This is evident in the implementation of the PSO solver, where the function addresses solving one solution at a time. Additionally, the function seamlessly manages arrays with multiple rows, a scenario encountered when simultaneously evaluating multiple solutions. This aspect will be further elaborated upon later in the context of pymoo and PySwarms solvers.

Listing 9.1 Solving the Michalewicz function using PSO

```
import numpy as np
import math
import matplotlib.pyplot as plt

def michalewicz_function(position):
    m = 10
    position = np.array(position)
    if len(position.shape) == 1:  #A
        position = position.reshape(1, -1)
    n = position.shape[1]
    j = np.arange(1, n + 1)
    s = np.sin(position) * np.power(np.sin((j * np.square(position)) /
➥ np.pi), 2 * m)
    return -np.sum(s, axis=1)
```

Reshape to a 2D array with a single row if the position is a size-1 array.

The Michalewicz formula

Let's now create a PSO solver from scratch. As a continuation of listing 9.1, we'll start by defining a particle class with position, velocity, and personal best value as follows:

```
class Particle:
    def __init__(self, position, velocity, pbest_position, pbest_value):
        self.position = position
        self.velocity = velocity
        self.pbest_position = pbest_position
        self.pbest_value = pbest_value
```

The fitness function to be minimized is the Michalewicz function in this example:

```
def fitness_function(position):
    return michalewicz_function(position)
```

We can now define the velocity update function following equation 9.1. The function takes three arguments—`particle`, which is an object representing the current particle; `gbest_position`, which is the global best position found by the swarm so far; and `options`, which is a dictionary containing the algorithm parameters (specifically, the inertia weight `w` and the cognitive and social acceleration coefficients `c1` and `c2`):

```
def update_velocity(particle, gbest_position, options):
    w = options['w']
    c1 = options['c1']
    c2 = options['c2']
    inertia = w * particle.velocity
    cognitive = c1 * np.random.rand() * (particle.pbest_position -
➥ particle.position)
    social = c2 * np.random.rand() * (gbest_position - particle.position)
    new_velocity = inertia + cognitive + social
    return new_velocity
```

The function computes the three components of the new velocity: the inertia component, the cognitive component, and the social component, as per equation 9.1. It returns the updated velocity as the sum of the three components.

We can now define the PSO solver function. This function takes four parameters as inputs—`swarm_size`, which is the size of the particle swarm; `iterations`, which is the maximum number of iterations to run the algorithm for; `bounds`, which is a list of tuples defining the lower and upper bounds of the search space for each dimension of the input vector; and `options`, which is a dictionary containing the algorithm parameters (such as the inertia weight and cognitive and social acceleration coefficients):

```
def pso(swarm_size, iterations, bounds, options):
    swarm = []
    for _ in range(swarm_size):
        position = np.array([np.random.uniform(low=low, high=high) for low,
➥ high in bounds])
        velocity = np.array([np.random.uniform(low=-abs(high-low),
➥ high=abs(high-low)) for low, high in bounds])
        pbest_position = position
        pbest_value = fitness_function(position)
        particle = Particle(position, velocity, pbest_position, pbest_value)
        swarm.append(particle)

    gbest_position = swarm[np.argmin([particle.pbest_value for particle in
➥ swarm])].pbest_position
    gbest_value = np.min([particle.pbest_value for particle in swarm])

    for _ in range(iterations):
        for _, particle in enumerate(swarm):
```

Initialize a random swarm.

Initialize the global best.

```
            particle.velocity = update_velocity(particle, gbest_position,
options)
            particle.position += particle.velocity

            particle.position = np.clip(particle.position, [low for low, high
➥ in bounds], [high for low, high in bounds])

            current_value = fitness_function(particle.position)
            if current_value < particle.pbest_value:
                particle.pbest_position = particle.position
                particle.pbest_value = current_value

            if current_value < gbest_value:
                gbest_position = particle.position
                gbest_value = current_value

            particle.position += particle.velocity

    return gbest_position, gbest_value
```

Update the velocity and position.

Apply the bounds.

Update the personal best (pbest).

Update the global best (gbest).

Update the position.

Return the global best position and corresponding value.

The function first initializes the particle swarm by randomly generating the initial positions and velocities for each particle within the bounds defined by `bounds`. It then evaluates the fitness function for each particle and updates its personal best position and value accordingly. The function then enters a loop, where it updates the velocity and position of each particle using the `update_velocity` function, which takes the global best position found so far as input. The function also applies bounds to the particle position and updates its personal best position and value. The function then updates the global best position and value based on the star topology. Other topologies, such as ring, Von Neumann, and wheel, can be found in the complete code of listing 9.1, available in the book's GitHub repository. Finally, the function returns the global best position and value found by the algorithm.

We can now use this PSO solver to minimize the Michalewicz function after we set up the problem and algorithm parameters as follows:

```
swarm_size = 50
iterations = 1000
options = {'w': 0.9, 'c1': 0.5, 'c2': 0.3}

dimension = 2
bounds = [(0, math.pi)] * dimension

best_position, best_value = pso(swarm_size, iterations, bounds, options)
```

PSO parameters

Dimension and domain of the Michalewicz function for each variable

Use the implemented PSO solver to solve the problem.

You can print the optimal solution and minimum value of the function after running PSO:

```
print(f"Optimal solution: {np.round(best_position,3)}")
print(f"Minimum value: {np.round(best_value,4)}")
print()
```

The output would be as follows:

```
Optimal solution: [2.183 1.57]
Minimum value: [-1.8013]
```

Compared to genetic algorithms, there are fewer Python libraries available for PSO. Pymoo provides a PSO implementation for continuous problems. As a continuation of listing 9.1, pymoo PSO can be used to solve the same problem as follows:

```
from pymoo.algorithms.soo.nonconvex.pso import PSO
from pymoo.core.problem import Problem
from pymoo.optimize import minimize

class MichalewiczFunction(Problem):
    def __init__(self):
        super().__init__(n_var=2,
                         n_obj=1,
                         n_constr=0,
                         xl=0,
                         xu=math.pi,
                         vtype=float)

    def _evaluate(self, x, out, *args, **kwargs):
        out["F"]= michalewicz_function(x)

problem = MichalewiczFunction()

algorithm = PSO(w=0.9, c1=2.0, c2=2.0)

res = minimize(problem,
               algorithm,
               seed=1,
               verbose=False)

print(f"Optimal solution: {np.round(res.X,3)}")
print(f"Minimum value: {np.round(res.F,4)}")
```

Define the problem.

The 2D Michalewicz function

Set the lower and upper bounds.

Evaluate the objective function.

Create a problem instance.

Define the solver with the parameters.

Apply PSO to solve the problem.

Print the optimal solution and minimum value of the function after running PSO.

This code produces the following output:

```
Optimal solution: [2.203 1.571]
Minimum value: [-1.8013]
```

PySwarms is another open source optimization library for Python that implements different variants of PSO. PySwarms can be used as follows to handle the problem:

```
!pip install pyswarms
import pyswarms as ps

dimension = 2
bounds = (np.zeros(dimension), np.pi * np.ones(dimension))

options = {'w': 0.9, 'c1': 0.5, 'c2': 0.3}

optimizer = ps.single.GlobalBestPSO(n_particles=100, dimensions=dimension,
➥ options=options, bounds=bounds)
```

Import the PSO solver from pyswarms.

Dimension of the Michalewicz function

Create bounds for the search space.

Set up the optimizer.

Create an instance of the optimizer.

Optimize the Michalewicz function.

```
cost, pos = optimizer.optimize(michalewicz_function, iters=1000)

print(f"Optimal solution: {np.round(pos,3)}")
print(f"Minimum value: {np.round(cost,4)}")
```

Print the optimal solution and minimum value of the function after running PSO.

This code produces the following output:

```
Optimal solution: [2.203 1.571]
Minimum value: -1.8013
```

The pyswarms.single package implements various techniques in continuous single-objective optimization. From this module, we used the global-best PSO (*gbest* PSO) algorithm in the previous code. You can experiment by replacing this solver with local-best.

Figure 9.6 shows the 3D landscape and 2D contours of the Michalewicz function, the optimal solution, and the solutions obtained by PSO, PSO Pymoo, and PSO PySwarms.

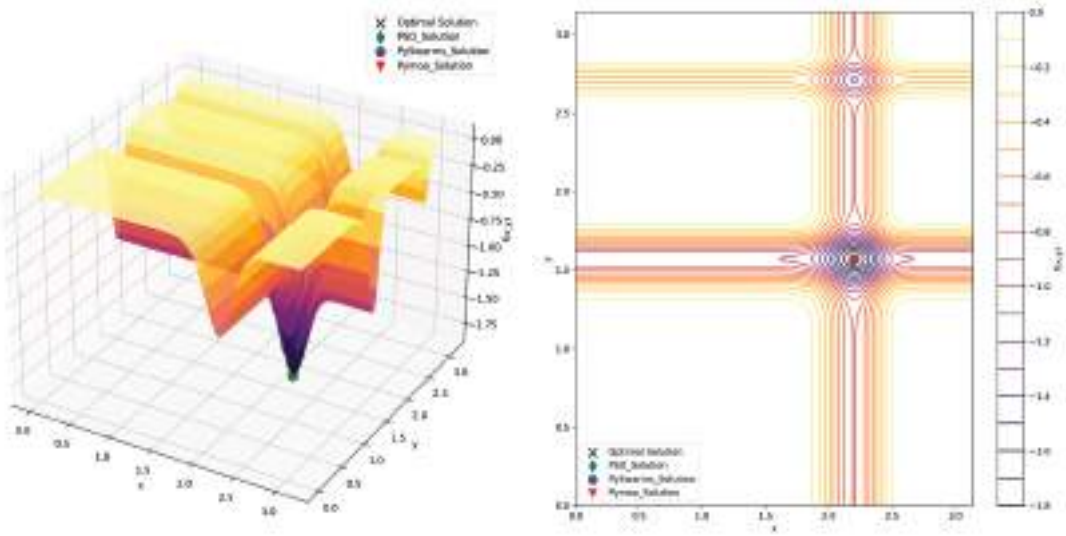


Figure 9.6 3D and 2D plots of the Michalewicz function. The three solutions are all very close to the optimal solution.

The three versions of PSO provide comparable results. However, PSO PySwarms and PSO pymoo are more stable, as they provide more consistent results each time you run the code. The PySwarms library is more comprehensive than pymoo for PSO, as it provides implementations of different variants and topologies of PSO, including discrete PSO, which will be explained in the next two sections.

9.3 Binary PSO

PSO was originally developed for problems that involve continuous-valued variables. However, many real-world problems are discrete or combinatorial in nature, such as TSP, task allocation, scheduling, assignment problems, and feature selection, among others. These types of problems involve searching through a finite set of possible solutions,

rather than searching through a continuous space. To handle these discrete problems, PSO variants have been developed, such as binary PSO and permutation-based PSO.

In *binary PSO* (BPSO), each particle represents a position in the binary space, where each element is either 0 or 1. The binary sequence is updated bit by bit based on its current value, the fitness-based value of that particular bit within the particle, and the best value of the same bit observed so far among its neighboring particles. This approach enables the search to be conducted in a binary space rather than a continuous space, which is well suited for problems where the variables are binary.

In BPSO, the velocity is defined in terms of the probability of the bit changing. To restrict the values of the velocity elements to the range of [0,1], the sigmoid function is used:

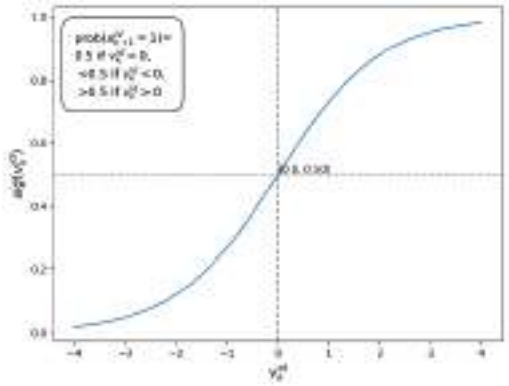
$$sig\left(v_k^{id}\right) = \frac{1}{1 + e^{-v_k^{id}}} \tag{9.6}$$

The position update equation then becomes

$$x_k^{id} = \begin{cases} 1 \text{ if } sig\left(v_k^{id}\right) > r \\ 0 \text{ otherwise} \end{cases} \tag{9.7}$$

where r is a randomly generated number in [0, 1]. Figure 9.7 shows the sigmoid function and the probability of the updated position to be 1. For example, if $v = 0.3$, this means that the probability that the updated position will be 1 is 30%, and the probability that it will be 0 is 70%.

Particle *i*						
Position	1	2	...	*d*	...	*n*
State	0	1	*	x_k^{id}	*	1



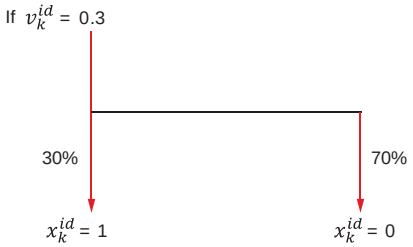


Figure 9.7 Position and velocity notations in binary PSO (BPSO). Each particle represents a position in the binary space. Velocity is defined in terms of the probability of the bit changing.

As you'll notice, the velocity components will remain as real-valued numbers using the original equation, but these values are then passed through the sigmoid function before updating the position vector. The following equations are the velocity update equations in BPSO:

$$v_{k+1}^{id} = v_k^{id} + \phi_1 \left(pbest_k^{id} - x_k^{id} \right) + \phi_2 \left(gbest_k^{d} + x_k^{id} \right) \quad \textbf{9.8}$$

$$sig\left(v_{k+1}^{id} \right) = \frac{1}{1 + e^{-v_{k+1}^{id}}} \quad \textbf{9.9}$$

The positions are updated according to the following equation:

$$x_{k+1}^{id} = \begin{cases} 1 \text{ if } sig\left(v_{k+1}^{id} \right) > r \\ 0 \text{ otherwise} \end{cases} \quad \textbf{9.10}$$

where

- ϕ_1 and ϕ_2 represent different random numbers drawn from uniform distributions. Sometimes these parameters are chosen from a uniform distribution 0–2, such that the sum of their two limits is 4.0.
- v_{k+1}^{id} is the probability that an individual i will choose 1 for the bit at the d^{th} site in the bit string.
- x_k^{id} is the current state of string i at bit d.
- v_k^{id} is a measure of the string's current probability to choose 1.
- $pbest_k^{id}$ is the best state found so far for bit d of individual i (i.e., a 1 or a 0).
- $gbest_k^{d}$ is 1 or 0 depending on what the value of bit d is in the best neighbor to date.

BPSO example

To illustrate how BPSO works, suppose we have a population of five binary particles, where each particle consists of 6 bits. Let's assume the particles are represented by the following binary strings: 101101, 110001, 011110, 100010, and 001011. We want to update particle 4 (represented by the binary string 100010) at bit 3 (which has a current value of 0). The current propensity (velocity) of this bit to be 1 is assumed to be 0.23. Additionally, we assume that the best value of this particle found so far is 101110, while the best value found by the entire population is 101111. Let's also assume that $\Phi_1 = 1.5$ and $\Phi_2 = 1.9$. Using equations 9.8 and 9.9, we can get the updated velocity of bit 3 in particle 4 as follows:

Particle 4: 100010, $v_k^{43} = 0.23$, $x_k^{43} = 0$, $pbest_k^{43} = 1$, $gbest_k^{3} = 1$, $\Phi_1 = 1.5$, $\Phi_2 = 1.9$

$v_{k+1}^{43} = 0.23 + 1.5(1-0) + 1.9(1-0) = 3.63$

$sig(v_{k+1}^{43}) = sig(3.63) = 1/(1 + e^{-3.63}) = 0.974$

(continued)

Generate a random number $r^{43} = 0.6$, and update the position using equation 9.10 as follows:

$x_{k+1}{}^{43} = 1$ as $sig(v_{k+1}{}^{43}) > r^{43}$

Updated particle 4: 100110

For more information about BPSO, see Kennedy and Eberhart's article "A discrete binary version of the particle swarm algorithm" [2].

9.4 Permutation-based PSO

Numerous efforts have been undertaken to employ PSO in solving permutation problems. The challenge of adapting PSO to tackle these problems arises from the fact that the notions of velocity and direction are not inherently applicable to permutation problems. To overcome this obstacle, arithmetic operations like addition and multiplication need to be redefined.

In M. Clerc's 2004 article, "Discrete particle swarm optimization, illustrated by the traveling salesman problem" [3], PSO was applied to solve the TSP. The position of a particle was the solution to a problem (the permutation of cities). The velocity of a particle was defined as the set of swaps to be performed on a particle. As you have seen, the right side of equation 9.1 contains three arithmetic operations: multiplication, subtraction, and addition. These operations are redefined for the new search space as follows:

- *Multiplication*—The velocity vector constrains a number of swaps between cities. Multiplying this vector by a constant c, results in another velocity vector with a different length, depending on the value of the constant. If $c = 0$, the length of the velocity vector (i.e., the included number of swaps) is set to 0. This means that no swap will be performed. If $c < 1$, the velocity is truncated. If $c > 1$, the velocity is augmented as illustrated in figure 9.8. Augmentation means adding a swap taken from the top of the current velocity vector to the end of the new velocity vector.

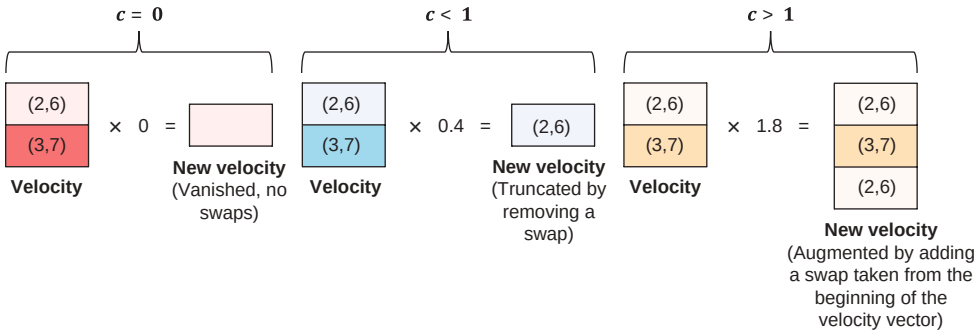


Figure 9.8 Redefined multiplication for permutation-based PSO

- *Subtraction*—Subtracting two positions should produce a velocity. This operation produces the sequence of swaps that could transform one position to the other. For example, let's consider an 8-city TSP. A candidate solution for this TSP is represented by a permutation such as [2, 4, 6, 1, 5, 3, 8, 7]. Figure 9.9 shows how a new velocity vector is produced by subtracting two positions.

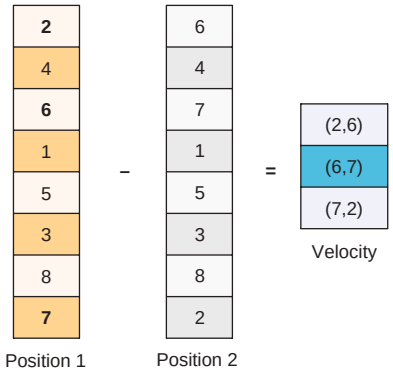


Figure 9.9 Redefined subtraction operation for permutation-based PSO

- *Addition*—The operation is performed by applying the sequence of swaps defined by the velocity to the position vector. Figure 9.10 shows how a new position (i.e., a new candidate solution) is generated by adding the velocity swap vector to the current position.

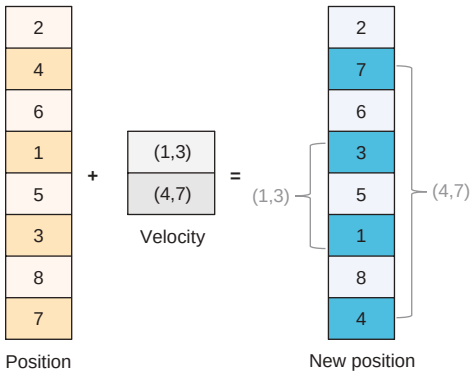


Figure 9.10 Redefined addition operation for permutation-based PSO

These redefined arithmetic operations allow us to update the velocity and position of PSO particles.

9.5 Adaptive PSO

The inertia, cognitive, and social components are the primary PSO parameters that can be used to achieve an equilibrium between exploration and exploitation during the optimization process. These three factors significantly influence the behavior of the algorithm, as discussed in the following subsections.

9.5.1 Inertia weight

The inertia parameter represents the tendency of a particle to maintain its current trajectory. By adjusting the inertia value, the algorithm can balance its focus on searching the solution space broadly (exploration) or homing in on the best solutions found thus far (exploitation). Large values of ω promote exploration, and small values promote exploitation, as illustrated in figure 9.11. Excessively small values may hinder the swarm's exploration capabilities. As the value of ω decreases, the influence of the cognitive and social components on position updates becomes more dominant.

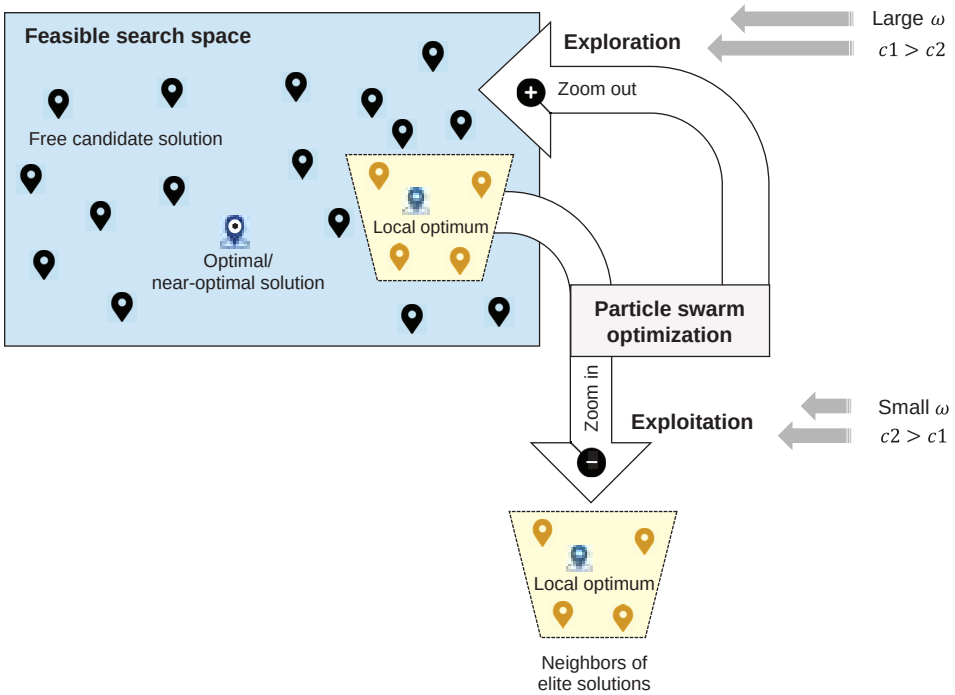


Figure 9.11 Effect of PSO parameters on the search behavior. Large inertia promotes exploration, and small values promote exploitation. *c*1 > *c*2 results in excessive wandering of individuals through the search space. In contrast, *c*2 > *c*1 may lead particles to rush prematurely toward a local optimum.

When $\omega > 1$, particle velocities tend to escalate over time, accelerating toward the maximum velocity (provided that velocity clamping is utilized), ultimately causing the swarm to diverge. In this scenario, particles struggle to alter their direction to return to promising regions. On the other hand, when $\omega < 1$, particles may gradually decelerate until their velocities approach 0, depending on the acceleration coefficients' values.

Velocity clamping can be considered by setting a maximum (and minimum) limit for the velocity. If the calculated velocity for a particle exceeds this limit, it is set to the maximum (or minimum) value. This prevents particles from wandering too far off in the problem space or getting stuck in a specific region in the search space.

The following methods can be used to update the inertia weight:

- *Random selection (RS)*—This involves selecting a different inertia weight in each iteration. The weight can be chosen from a distribution with a mean and standard deviation of your choice, but it's important to ensure that the swarm still converges despite the randomness. The following formula can be used:

$$\omega_t = 0.5 + \frac{rand(.)}{2} \tag{9.11}$$

  where $rand(.)$ is a uniformly distributed random number within the range [0,1]. Therefore, the mean value of the inertia weight is 0.75.

- *Linear time varying (LTV)*—This involves gradually decreasing the value of ω from a starting high value of ω_{max} to a final low value of ω_{min} following this equation:

$$\omega_t = (\omega_{max} - \omega_{min}) \times \frac{(t_{max} - t)}{t_{max}} + \omega_{min} \tag{9.12}$$

  where t_{max} is the number of iterations, t is the current iteration, and ω_t is the value of the inertia weight in the t^{th} iteration. Typically, the convention is to set ω_{max} and ω_{min} to 0.9 and 0.4 respectively.

- *Nonlinear time varying (NLTV)*—This approach also involves decreasing the inertia weight from an initial high value, but this decrement can be nonlinear, as shown in the following equation:

$$\omega_{t+1} = (\omega_t - \omega_{min}) \times \frac{(t_{max} - t)}{t_{max} + \omega_{min}} + \omega_{min} \tag{9.13}$$

  where $\omega_{t=0} = 0.9$ is the initial choice of ω. By allowing more time to fall off toward the lower end of the dynamic range, NLTV can enhance local search or exploitation.

Figure 9.12 shows these three update methods.

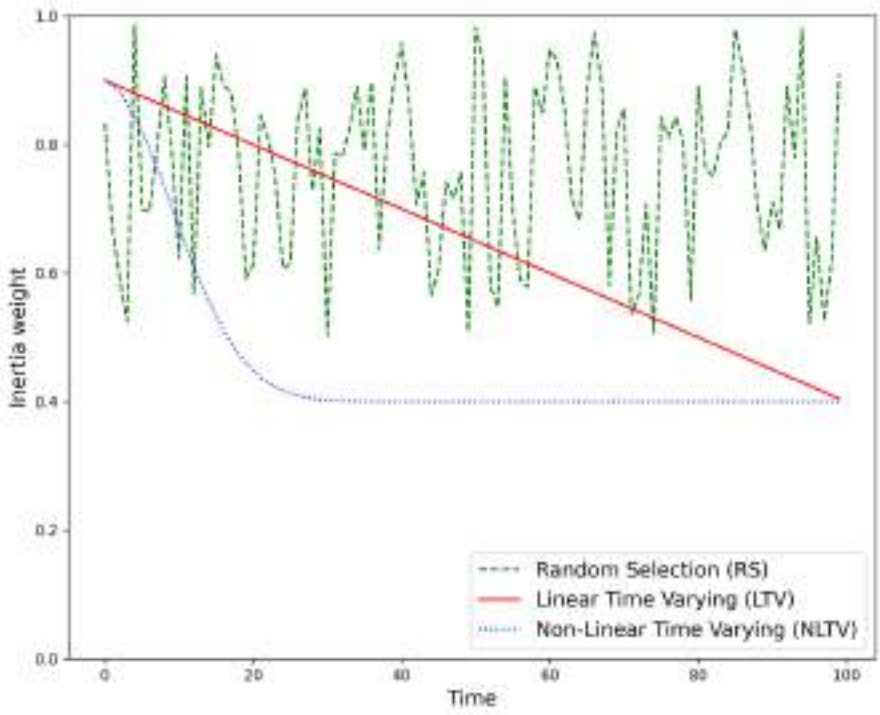


Figure 9.12 Different inertia weight update methods

As you can see, in random selection, a different inertia weight is randomly selected in each iteration. The mean value of the inertia weight is 0.75. LTV linearly decreases the inertia weight. In NLTV, the inertia weight decrement is more gradual than in LTV. In summary, the inertia weight plays a crucial role in the convergence speed and solution quality of the PSO algorithm. A high inertia weight promotes exploration, while a low inertia weight encourages exploitation.

9.5.2 Cognitive and social components

The cognitive component $c1$ is a parameter associated with a particle's individual learning capability, where the particle is influenced by its own experiences. The social component $c2$ is a parameter linked to the collective learning capability of all particles within the swarm. It represents the degree to which a particle is influenced by the best solutions found by its neighbors. If $c1 > c2$, the algorithm will show exploratory behavior, and if $c2 > c1$, the algorithm will tend to exploit the local search space, as illustrated in figure 9.11. Setting $c1 = 0$ reduces the velocity model to a social-only model or selfless model (the particles are all attracted to *nbest*). On the other hand, setting $c2 = 0$ reduces it to a cognition-only model (particles are independent, as in the case of the hill climbing algorithm).

Typically, $c1$ and $c2$ are kept constant in PSO. Empirically, the sum of $c1$ and $c2$ should be less than or equal to 4, and any significant deviations from this may result in divergent behavior. In adaptive PSO, it is advisable to gradually decrease the value of $c1$ over time and concurrently increase the value of $c2$ using linear formulas [4], as follows:

$$c1_t = \left(\frac{(c1_{max} - c1_{min}) \times (t_{max} - t)}{t_{max}} \right) + c1_{min} \quad \textbf{9.14}$$

$$c2_t = c2_{max} - \left(\frac{(c2_{max} - c2_{min}) \times (t_{max} - t)}{t_{max}} \right) \quad \textbf{9.15}$$

where t is the iteration index, $c1_{max}$ and $c2_{max}$ are the maximum cognitive and social parameters respectively, $c1_{min}$ and $c2_{min}$ are the minimum cognitive and social parameters respectively, and t_{max} is the maximum iteration.

Figure 9.13 shows the linearly changing $c1$ and $c2$. As you can see, we start with $c1 > c2$ to favor exploration. As the search progresses, $c2$ starts to be higher than $c1$ in order to favor exploitation.

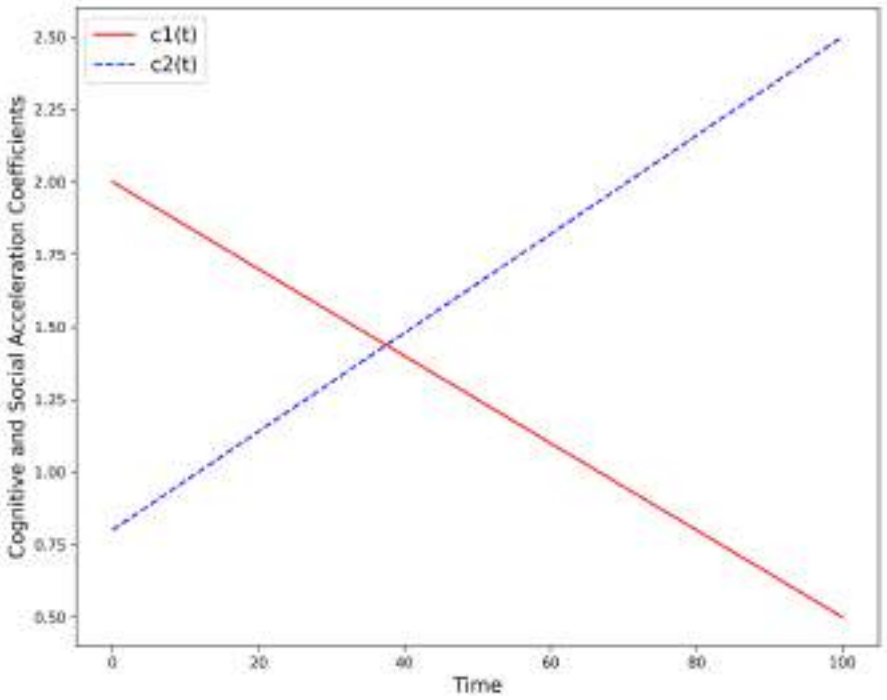


Figure 9.13 Cognitive and social acceleration coefficient updates. c1>c2 results in more exploration, while c2>c1 may lead to more exploitation.

Let's now see how we can use PSO to handle continuous and discrete optimization problems.

9.6 Solving the traveling salesman problem

In the previous chapter, you saw how to solve the TSP for 20 major cities in the United States, starting from New York City, using a genetic algorithm. Let's now solve the same problem using PSO, as shown in the next listing. We'll start by defining the latitude and longitude for the twenty US cities and computing the inter-city distances between them.

Listing 9.2 Solving TSP using PSO

```
import numpy as np
import pandas as pd
from collections import defaultdict
from haversine import haversine
import networkx as nx
import matplotlib.pyplot as plt
import pyswarms as ps

cities = {
    'New York City': (40.72, -74.00),
    'Philadelphia': (39.95, -75.17),
    'Baltimore': (39.28, -76.62),
    'Charlotte': (35.23, -80.85),
    'Memphis': (35.12, -89.97),
    'Jacksonville': (30.32, -81.70),
    'Houston': (29.77, -95.38),
    'Austin': (30.27, -97.77),
    'San Antonio': (29.53, -98.47),
    'Fort Worth': (32.75, -97.33),
    'Dallas': (32.78, -96.80),
    'San Diego': (32.78, -117.15),
    'Los Angeles': (34.05, -118.25),
    'San Jose': (37.30, -121.87),
    'San Francisco': (37.78, -122.42),
    'Indianapolis': (39.78, -86.15),
    'Phoenix': (33.45, -112.07),
    'Columbus': (39.98, -82.98),
    'Chicago': (41.88, -87.63),
    'Detroit': (42.33, -83.05)
}

distance_matrix = defaultdict(dict)
for ka, va in cities.items():
    for kb, vb in cities.items():
        distance_matrix[ka][kb] = 0.0 if kb == ka
➥ else haversine((va[0],va[1]), (vb[0], vb[1]))

distances = pd.DataFrame(distance_matrix)
distance=distances.values
city_names=list(distances.columns)
```

Define the latitude and longitude for twenty major US cities.

Create a haversine distance matrix based on the latitude and longitude coordinates.

Convert the distance dictionary into a dataframe with distances as values and city names as headers.

Next, we can count the number of cities and set up the integer bounds of the decision variables, which represent the order in which the cities are visited. The first function, `tsp_distance`, takes two arguments: `position` and `distance`. `position` is a 1D array that represents the order in which the cities are visited. `distance` is a 2D array that contains the distances between all pairs of cities. The function first defines `tour` as a permutation of the indices that represent the order of visiting the cities. It then calculates the total distance of the tour by summing the distances between adjacent cities as well as the distance between the last city in the tour and the starting city.

The second function, `tsp_cost`, takes two arguments: `x` and `distance`. `x` is a 2D array that contains the decision variables for the TSP problem, with each row representing a different particle in the swarm. `distance` is a 2D array that contains the distances between all pairs of cities. The function calculates the cost of each particle by calling the `tsp_distance` function on each row of `x` and returns a list of the costs:

Define the TSP problem as a permutation optimization problem with integer bounds.

Define the TSP distance function.

Convert the permutation to a TSP tour.

Compute the total distance of the tour from New York City as the first city, and add the distance from the last city back to New York City.

Compute and return the cost of each particle in the swarm.

```
n_cities = len(city_names)
bounds = (np.zeros(n_cities), np.ones(n_cities)*(n_cities-1))

def tsp_distance(position, distance):
    tour = np.argsort(position)
    total_distance = distance[0, tour[0]]
    for i in range(n_cities-1):
        total_distance += distance[tour[i], tour[i+1]]
    total_distance += distance[tour[-1], 0]
    return total_distance

def tsp_cost(x, distance):
    n_particles = x.shape[0]
    cost=0
    cost = [tsp_distance(x[i], distance) for i in range(n_particles)]
    return cost
```

As a continuation of listing 9.2, the following code sets the parameters for the PSO optimizer. `options` is a dictionary that contains the values for the inertia weight (`w`), cognitive (`c1`) and social (`c2`) acceleration coefficients, number of neighbors to consider (`k`), and the p-value for the Minkowski distance (`p`). `n_particles` represents the number of particles used in the optimization, and `dimensions` represents the number of decision variables, which is equal to the number of cities in the TSP problem. The best solution found by the optimizer is converted to a TSP tour by sorting the indices of the solution in ascending order and using them to index the `city_names` list in the same order. This creates a list of city names in the order that they are visited in the best tour. We then print the best route and its length:

Set up the PSO parameters.

Instantiate the PSO optimizer.

```
options = {'w': 0.79, 'c1': 2.05, 'c2': 2.05, 'k': 10, 'p': 2}
optimizer = ps.discrete.BinaryPSO(n_particles=100, dimensions=n_cities,
➥ options=options)
```

```
cost, solution = optimizer.optimize(tsp_cost, iters=150, verbose=True,
➥ distance=distance)
```

Solve the problem.

```
tour = np.argsort(solution)
city_names_tour = [city_names[i] for i in tour]
```

Convert the best solution to a TSP tour.

```
Route = " → ".join(city_names_tour)
print("Route:", Route)
print("Route length:", np.round(cost, 3))
```

Print the best route and its length.

Listing 9.2 produces the following output:

```
Route: New York City → Columbus → Indianapolis → Memphis → San Francisco
→ San Jose → Los Angeles → San Diego → Phoenix → Dallas → Fort Worth →
San Antonio → Austin → Houston → Jacksonville → Charlotte → Baltimore →
Philadelphia → Chicago → Detroit
Route length: 12781.892
```

Figure 9.14 shows the obtained route. The complete version of listing 9.2 is available in the book's GitHub repo, and it shows the steps for visualizing the route as a NetworkX graph.

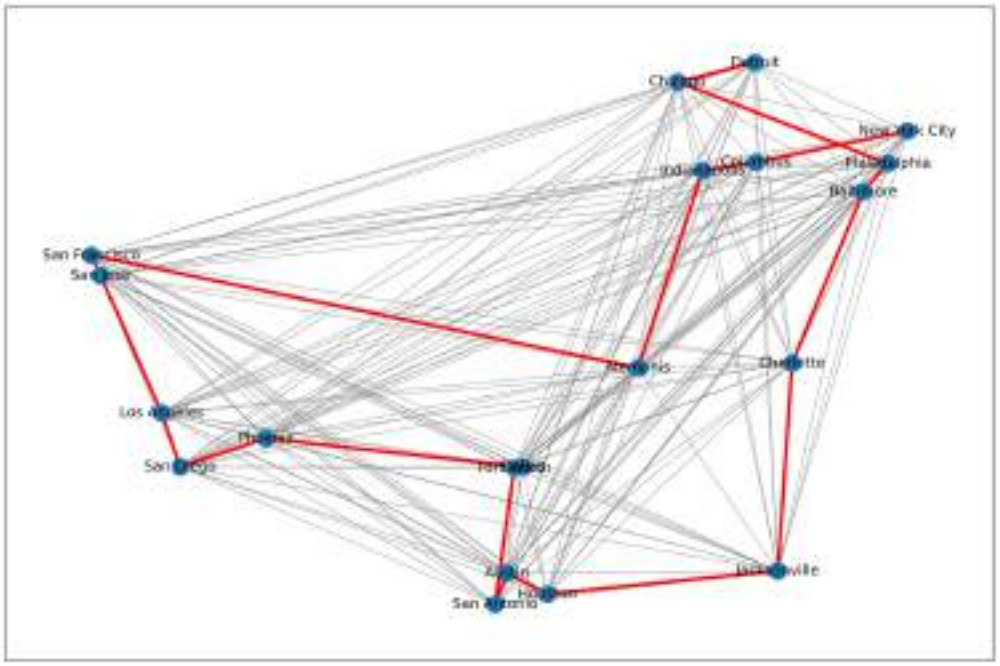


Figure 9.14 PSO solution for the 20-city TSP

Feel free to adjust the code according to your needs by modifying elements such as the problem data, the initial city, or the parameters of the algorithm.

9.7 Neural network training using PSO

Machine learning (ML) is a subfield of artificial intelligence (AI) that endows an artificial system or process with the ability to learn from experience and observation without being explicitly programmed. Many ML approaches have been and are still being proposed, and more details about ML will be provided in chapter 11. For now, let's consider neural networks, which are one of the most used and successful statistical ML approaches. The artificial neural network (ANN or NN) approach is inspired by the biological brain and can be considered a highly simplified computational model, as NN is very far from matching a brain's complexity. NN is at the heart of deep learning models that nowadays form the basis of many successful applications that touch everybody's life, such as text, audio, image, and video generation, voice assistants, and recommendation engines, to name just a few.

The human brain

Aristotle (384-322 BC) wrote, "Of all the animals, man has the largest brain in proportion to his size." The human brain is composed of an average of 86 billon interconnected nerve cells or *neurons*. Each biological neuron is connected to several thousand other neurons. It is extremely energy efficient, as it can perform the equivalent of an exaflop (a billion billion mathematical operations per second) with just 20 watts of power.

For simplicity, consider ML as glorified curve fitting, which intends to find a mapping function between independent and dependent variables. For example, suppose a vision-based object recognition model takes as input a digital image taken by the front camera of a vehicle—the output would be recognized objects, such as cars, pedestrians, cyclists, lanes, traffic lights, etc. In fact, ML shares the same ingredients as curve fitting in terms of model, scoring criteria, and search strategy. However, ML approaches, such as NNs, are a way to create functions that no human could write. They tend to create nonlinear, nonmonotonic, nonpolynomial, and even noncontinuous functions that approximate the relationship between independent and dependent variables in a data set.

An NN is a massively parallel adaptive network of simple nonlinear computing elements called neurons that are arranged in input, hidden, and output layers. Each node, or artificial neuron, connects to another and has an associated weight and threshold allowing the node to simulate a neuron firing. Each individual node has its own linear regression model, composed of input data, a bias, a threshold, and an output, as illustrated in figure 9.15. A neuron k can be described with the following equation:

$$z_k = \sum_{i=1}^{n} \omega_{ki} x_i + b \tag{9.16}$$

Its output is

$$y_k = \phi\left(z_k - \theta_k\right) \tag{9.17}$$

where x_i are the inputs, ω_{ki} are the weights, b is the bias term that defines the ability to fire in the absence of external input to the node, and φ is the activation function. This activation function makes the neuron fire the output when the input z_k reaches a threshold θ_k. There are different forms of activation functions (aka squashing functions) such as sign, step, tanh, arctan, s-shaped sigmoid (aka logistic), softmax, radial basis function, and rectified linear unit (ReLU).

As in the case of curve fitting, a scoring criterion or cost function is used to estimate deviation between the estimated values and the actual values. In this context, training an NN is fundamentally an optimization problem. The goal of training is to find the optimal parameters (weights and biases) that minimize the difference between the network's output and the expected output. This difference is often quantified using a loss or cost function, such as these:

- *Mean squared error (MSE)*—MSE is often used in regression problems. It calculates the square of the difference between the predicted and actual values and then averages these across the dataset. This function heavily penalizes large errors.
- *Cross-entropy loss*—Cross-entropy loss is typically used for binary and multiclass classification problems. It measures the dissimilarity between the predicted probability distribution and the actual distribution. In other words, it compares the model's confidence in its prediction with the actual outcome.
- *Negative log likelihood (NLL)*—NLL is another loss function in multiclass classification. If y is the true label and $p(y)$ is the predicted probability of that label, the negative log likelihood is defined as $-\log(p(y))$. The log function transforms the probabilities, which range between 0 and 1, to a scale that ranges from positive infinity to 0. When the predicted probability for the correct class is high (close to 1), the log value is closer to 0, but as the predicted probability for the correct class decreases, the log value increases toward infinity. Negating the log value thus gives a quantity that is minimized when the predicted probability for the correct class is maximized.

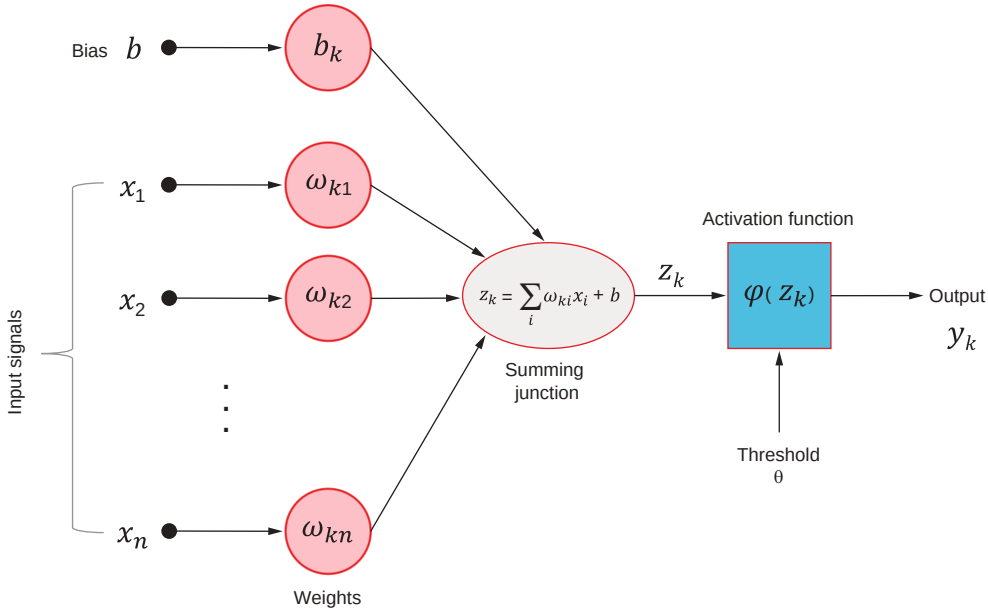


Figure 9.15 Neural network node demonstration

Training an NN involves the following steps:

- *Initialization*—Before training starts, the weights and biases in the network are typically initialized with small random numbers.
- *Feedforward*—In this stage, the input is passed through the network to produce an output. This output is generated by performing computations on the inputs using the initial or current weights, bias, and activation function transformation. The output of one layer becomes the input to the next layer until the final output is produced.
- *Error calculation*—After the feedforward stage, the output is compared with the desired output to calculate the error using a loss function. This function quantifies how far the network's predictions are from the actual values.
- *Backpropagation*—The calculated error is then propagated back through the network, starting from the output layer and moving back toward the input layer. This process computes the gradient or derivative of the loss function with respect to the weights and biases in the network.

- *Weight adjustment*—In this final stage, the weights of the network are updated in an effort to reduce the error. This is typically done using a technique called *gradient descent*. The weights are adjusted in the direction that most decreases the error, as determined by the gradients calculated during backpropagation.

By repeating these steps for many iterations (or epochs), the network gradually learns to produce outputs that are closer to the desired ones, thus "learning" from the input data.

Now that you have a basic understanding of NNs, let's train a simple NN using PSO following a supervised learning approach. During supervised training, the NN learns by initially processing a labeled dataset. By training on a labeled dataset, the network can subsequently predict labels for a new, unlabeled data set during the inferencing stage, after training.

For this example, we will use the Penguins dataset. This is a popular dataset in the data science community, containing information on the size, sex, and species of penguins. The dataset consists of 344 observations collected from three islands in the Palmer Archipelago, Antarctica. It includes the following seven variables:

- `species`—The species of penguin (Adelie, Chinstrap, or Gentoo)
- `island`—The island where the penguin was observed (Biscoe, Dream, or Torgersen)
- `bill_length_mm`—The length of the penguin's bill in millimeters
- `bill_depth_mm`—The depth of the penguin's bill in millimeters
- `flipper_length_mm`—The length of the penguin's flipper in millimeters
- `body_mass_g`—The mass of the penguin's body in grams
- `sex`—The sex of the penguin (male or female)

Our simple NN, described in the PySwarms use cases, has the following characteristics:

- *Input layer size*—4
- *Hidden layer size*—10 (activation function: $\tanh(x)$). The hyperbolic tangent activation function (aka Tanh, tanh, or TanH) maps input values to be between –1 and 1, and it's used to introduce nonlinearity in NNs. Remember that a sigmoid function maps input values to be between 0 and 1. The tanh function is centered at 0, which helps mitigate the vanishing gradient problem, compared to the sigmoid function. However, both tanh and sigmoid activations suffer from the vanishing gradient problem to some degree. Alternatives like rectified linear unit (ReLU) and its variants are often preferred.
- *Output layer size*—3 (activation function: $\text{softmax}(x)$). Softmax is a generalization of the sigmoid function. This function takes as input the *logits* that represent unnormalized outputs of the last layer of the network before they are

transformed into probabilities by applying a softmax function. These logits can be interpreted as a measure of the "evidence" that a certain input belongs to a particular class. The higher the logit value for a particular class, the more likely it is that the input belongs to that class.

The following listing shows the steps for training this simple NN using PSO. We start by importing the libraries we'll need and reading the penguin dataset.

Listing 9.3 Neural network training using PSO

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.decomposition import PCA
import pyswarms as ps

penguins = sns.load_dataset('penguins')
penguins.head()
```

Required for loading the dataset

Required for target label encoding

Required for dimensionality reduction

Load the Penguins dataset.

Show the dataset rows and columns.

This produces the output shown in figure 9.16.

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female

Figure 9.16 Penguins dataset

As a continuation of listing 9.3, we can visualize this dataset using the seaborn library as follows:

```
plt.figure(figsize=(8, 6))
sns.scatterplot(data=penguins, x='bill_length_mm', y='body_mass_g',
➥ hue='species', style="species")
plt.title('Bill Length vs. Body Mass by Species')
plt.show()
```

The output is shown in figure 9.17.

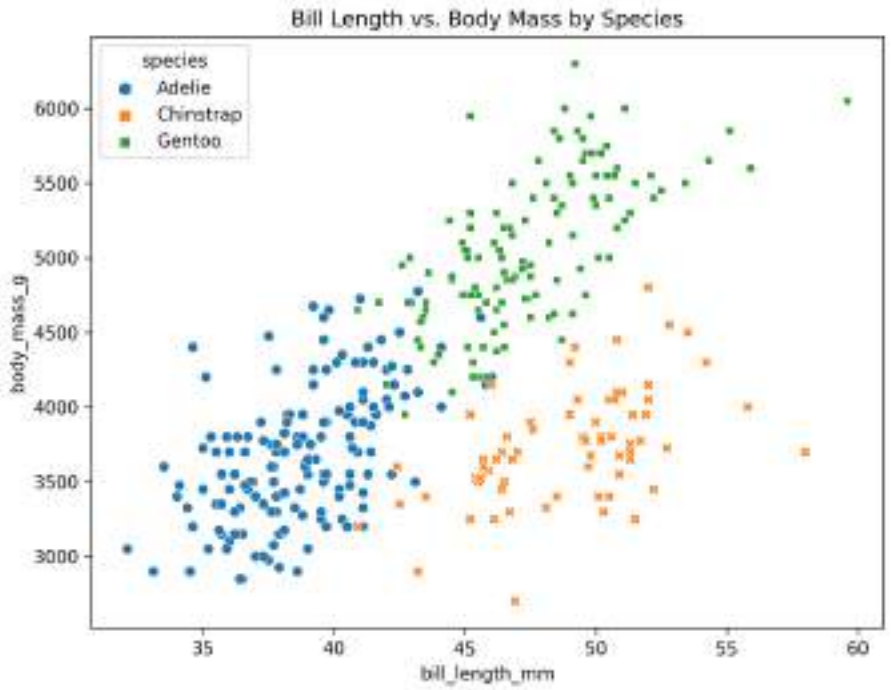


Figure 9.17 Bill length vs. body mass by species in the Penguins dataset

Next, we define a `logits_function` to take in a vector `p` of parameters for the NN and return the logits (pre-activation values) for the final layer of the network. As illustrated in figure 9.18, this function starts by extracting the weights and biases for the first and second layers of the network from the parameter vector p using indexing and reshaping operations. Then, the function performs forward propagation by computing the pre-activation value z^1 in the first layer as the dot product of the input data X and the first set of weights W^1, plus the bias term b^1. It then applies the tanh activation function to z^1 to obtain the activation value a^1 in the first layer. Finally, the function computes the pre-activation value for the second layer by taking the dot product of a^1 and the second set of weights W^2 and adding the bias term b^2. The resulting values are returned as the logits from the final layer of the network:

```
def logits_function(p):

    W1 = p[0: n_inputs * n_hidden].reshape((n_inputs, n_hidden))
    b1 = p[n_inputs * n_hidden: (n_inputs + 1) * n_hidden].reshape((n_hidden,))
    W2 = p[(n_inputs +1) * n_hidden: -n_classes].reshape((n_hidden, n_classes))
    b2 = p[-n_classes:].reshape((n_classes,))

    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
```

Extracting the weights of the first layer

Extracting the weights of the second layer

Extracting the biases of the second layer

Calculate the pre-activation value in the first layer .

```
    logits = a1.dot(W2) + b2
    return logits
```

Calculate and return the logits.

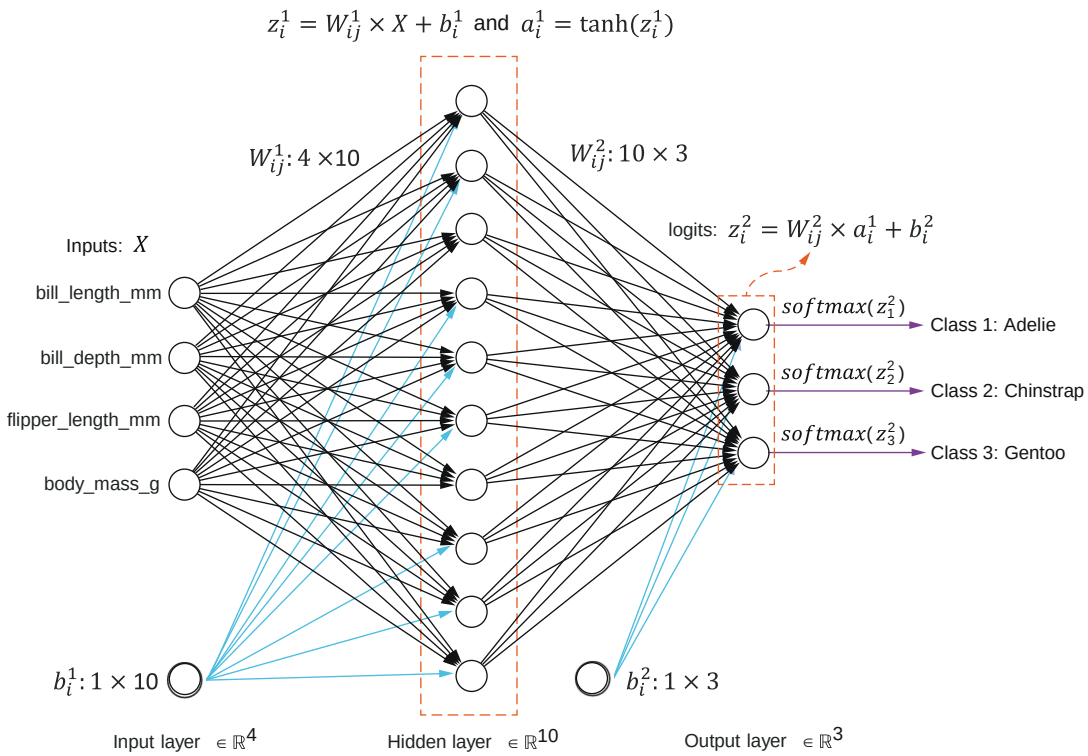


Figure 9.18 NN layers

Next, we define the `forward_prop` function to perform a forward pass through an NN with two layers. This computes the softmax probabilities and negative log likelihood loss for the output, given a set of parameters `params`. The function first calls the `logits_function` to obtain the logits for the final layer of the network, given the parameters `params`. Then the function applies the softmax function to the logits using the `np.exp` function and normalizes the resulting values by dividing by the sum of the exponentiated logits for each sample, using the `np.sum` function with the `axis=1` argument. This gives a probability distribution over the classes for each sample. The function then computes the negative log likelihood loss by taking the negative log of the probability of the correct class for each sample, which is obtained by indexing the `probs` array using the `y` variable, which contains the true class labels. The `np.sum` function is used to compute the sum of these negative log probabilities across all samples, and the result is divided by the total number of samples to obtain the average loss per sample. Finally, the function returns the computed loss:

```
def forward_prop(params):
    logits = logits_function(params)

    exp_scores = np.exp(logits)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    correct_logprobs = -np.log(probs[range(num_samples), y])
    loss = np.sum(correct_logprobs) / num_samples

    return loss
```

Obtain the logits for the softmax.

Compute the negative log likelihood.

Apply softmax to calculate the probability distribution over the classes.

Compute and return the loss.

To perform forward propagation over the whole swarm of particles, we define the following `particle_loss()` function. This function computes the loss for each particle in a PSO swarm, given its position in the search space. It is worth noting that each position represents the NN parameters (w1,b1,w2,b2) with `dimension` calculated as follows:

```
dimension = (n_inputs * n_hidden) + (n_hidden * n_classes) + n_hidden +
      n_classes = 4 * 10 + 10 * 3 + 10 * 3 + 1 * 10 + 1 * 3 = 83.
```

An example of a candidate setting of NN parameters (i.e., a *position* in PSO terminology) is given here:

```
[ 3.65105185e-01 -9.57472869e-02  4.99475198e-01  2.33703047e-01
  5.56295931e-01  6.95323783e-01  8.76045204e-02  5.52892675e-01
  3.33363337e-01  5.60680304e-01  3.24233602e-01  3.40402243e-01
  2.28940991e-01  6.47396295e-01  2.49476898e-01 -2.15041386e-01
  6.61749764e-01  4.50805880e-01  7.31521923e-01  4.55724886e-01
  5.81614992e-01  4.21303249e-01  3.10417945e-01  2.80091464e-01
  3.63352355e-01  7.21593713e-01  4.11009136e-01  3.50489680e-01
  6.82325768e-01  3.60945155e-01  3.34258781e-01  5.53845122e-01
  5.39748679e-01  8.45310205e-01  7.38728229e-01  5.44408893e-01
  4.22464042e-01  4.45099192e-01  4.36914661e-01 -2.40298612e-02
  4.68795601e-01  4.58388074e-01  2.29566792e-01  5.18783606e-01
  1.21226363e-01  2.80730816e-01  4.13249634e-01  1.91229505e-01
  6.30829496e-01 -4.52777424e-01  1.62066215e-01  3.07603861e-01
  1.54565454e-01  5.39974173e-01  4.48241886e-01 -2.81999490e-04
  2.93907050e-01  2.58571312e-01  7.87784363e-01  5.06092352e-01
  1.85010537e-01  8.06641243e-01  8.30985197e-01  4.06314407e-01
  2.20795704e-01  3.25405213e-01  6.02993839e-01  4.21051295e-01
  5.24352428e-01  2.49710316e-01  4.99212007e-01  4.48000964e-01
  4.90888329e-01  3.94908331e-01  6.35997377e-01  5.91192453e-01
  6.16639300e-01  6.85748919e-01  5.40805197e-01 -1.51195135e+00
  3.21751027e-01  3.93555680e-01  5.23679003e-01]
```

The PSO algorithm can then use these loss values to update the positions of the particles and search for the optimal set of parameters for the NN.

```
def particle_loss(x):
    n_particles = x.shape[0]
    j = [forward_prop(x[i]) for i in range(n_particles)]
    return np.array(j)
```

Determine the number of particles.

Compute and return the loss for each particle.

The last function we need is `predict`, which uses the NN parameters corresponding to the positions of particles in a PSO swarm to predict the class labels for each sample in the dataset. This function first calls `logits_function` to obtain the logits for the final layer of the network, given the positions `pos` of the particles in the search space. Then the function computes the predicted class labels by taking the `argmax` of the logits across the columns (i.e., along the second axis or `axis=1`), using the `np.argmax` function. This gives the index of the class with the highest probability for each sample. Finally, the function returns the predicted class labels as a numpy array `y_pred`:

Compute and return the predicted class labels.

```
def predict(pos):
    logits = logits_function(pos)
    y_pred = np.argmax(logits, axis=1)
    return y_pred
```

Obtain logits for the final layer of the network.

We can now train the NN using different PSOs available in PySwarms. The code starts by setting up several training samples, inputs, and the number of hidden layers and outputs. The dimensions are then computed based on the number of inputs, hidden nodes, and output classes. Three variants of PSO are defined: `globalBest`, `localBest`, and `binaryPSO`. The PSO hyperparameters are set using a dictionary called `options`. These hyperparameters include the inertia weight `w`, the cognitive parameter `c1`, the social parameter `c2`, the number of neighbors to be considered `k`, and the Minkowski distance parameter `p` (`p=1` is the sum-of-absolute values [or the L1 distance], while `p=2` is the Euclidean [or L2] distance):

Get the feature vector.

```
X = penguins[['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm',
➥ 'body_mass_g']].to_numpy()
num_samples = X.shape[0]
n_inputs = X.shape[1]
n_hidden = 10
n_classes = len(np.unique(y))

dimensions = (n_inputs * n_hidden) + (n_hidden * n_classes) + n_hidden + n_
classes

PSO_varaints = ['globalBest', 'localBest', 'binaryPSO']

options = {'w':0.79, 'c1': 0.9, 'c2': 0.5, 'k': 8, 'p': 2}

for algorithm in PSO_varaints:
    if algorithm == 'globalBest':

        optimizer = ps.single.GlobalBestPSO(n_particles=150,
➥ dimensions=dimensions, options=options)
        cost, pos = optimizer.optimize(particle_loss, iters=2000)
        print("#"*30)
        print(f"PSO varaints: {algorithm}")
        print(f"Best average accuracy: {100*round((predict(pos) ==
y).mean(),3)} %")
        print()
```

Set up the number of training samples, inputs, hidden layers, and outputs.

Set up the dimensions of the problem.

Define the variants of PSO.

Set up the PSO hyperparameters.

Train the NN using different variants of PSO, and print the best accuracy.

```
    elif algorithm == 'localBest':
        optimizer = ps.single.LocalBestPSO(n_particles=150,
➥ dimensions=dimensions, options=options)
        cost, pos = optimizer.optimize(particle_loss, iters=2000)
        print("#"*30)
        print(f"PSO varaints: {algorithm}")
        print(f"Best average accuracy: {100*round((predict(pos) ==
y).mean(),3)} %")
        print()
    elif algorithm == 'binaryPSO':
        optimizer = ps.discrete.BinaryPSO(n_particles=150,
➥ dimensions=dimensions, options=options)
        cost, pos = optimizer.optimize(particle_loss, iters=2000)
        print("#"*30)
        print(f"PSO varaints: {algorithm}")
        print(f"Best average accuracy: {100*round((predict(pos) ==
y).mean(),3)} %")
        print()
```

The code then trains an NN, using each variant of PSO in turn, by creating an instance of the PSO optimizer and calling the `optimize` method, passing in the loss function and the number of iterations to run, `iters`. The best loss and particle position found by the optimizer are stored in `cost` and `pos` respectively. The code then prints the variant of PSO used along with the best accuracy that was obtained by using the corresponding particle position to make a prediction and comparing it to the true class label `y`.

Running the complete listing produces the following output:

```
##############################
PSO variant: globalBest
Best average accuracy: 99.1 %
##############################
PSO variant: localBest
Best average accuracy: 69.1 %
##############################
PSO variant: binaryPSO
Best average accuracy: 43.8 %
```

As you can see, `globalBest` PSO is the most efficient PSO variant for training this NN. Binary PSO does not match with the continuous nature of the NN parameters.

You can experiment with the code by changing the problem and algorithm parameters. For example, you could use a reduced feature set such as `bill_length_mm` and `flipper_length_mm` instead of the four features used in this code. You could also change the algorithm parameters and apply velocity clamping. `velocity clamp` is a parameter enabled in PySwarms to set the limits for velocity clamping. It's a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity.

In the next chapter, you will be introduced to ant colony optimization (ACO) and artificial bee colony (ABC) as other effective optimization algorithms inspired by swarm intelligence.

Summary

- PSO employs a stochastic approach that utilizes the collective intelligence and movement of a swarm of particles. It is based on the idea of social interaction, which allows for efficient problem-solving.
- The fundamental principle of PSO is to guide the swarm toward the best position in the search space while also remembering each particle's own best-known position, as well as the global best-known position of the swarm.
- PSO is guided by a straightforward principle: emulate the success of neighboring individuals.
- Although PSO was initially designed for solving problems with continuous variables, many real-world problems involve discrete or combinatorial variables. In these problems, the search space is finite, and the algorithm needs to search through a set of discrete solutions. To address these types of problems, different variants of PSO have been developed, such as binary PSO (BPSO) and permutation-based PSO.
- By carefully tuning inertia weight and cognitive and social acceleration coefficients, PSO can effectively balance exploration and exploitation.