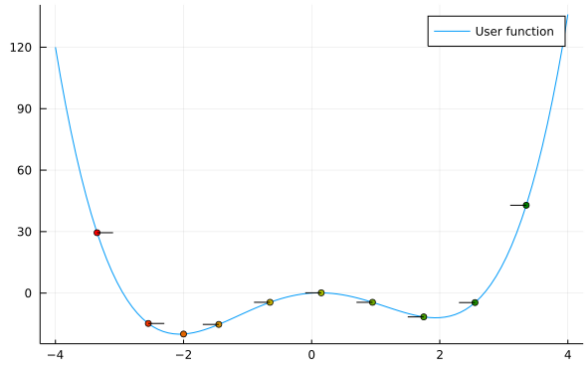
Particle swarm optimization to the MLJ machine learning toolbox

Google Summer of Code 2021 Proposal

Quazi Irfan

Mentors: Anthony Blaom, Sebastian Vollmer



MS Student in Statistics
South Dakota State University
Contact: 704 12th Ave, Apt 3
Brookings, South Dakota, 57006
Email: quazirfan@gmail.com;
quazi.irfan@jacks.sdstate.edu
Websites: github.com/quazi-irfan/
Slack nick: Quazi Irfan
Telephone number: +1 386 334 4792

Google Summer of Code '21
Organization: Julia Language
Package: MLJ / MLJ Tuning

[1] Please use Adobe Acrobat Reader to open this PDF as all plots are animated.
[2] No code is included in this PDF. All code are hosted at my Github repo

Contents

1 Introduction

MLJ(Machine Learning in Julia) is a toolbox written in Julia that provided uniform interface for fitting, evaluating, tuning and benchmark models. This toolbox uses scientific types to ease handling different data type and allows model composition using pipelining.

Each MLJ model can be tuned with a set of parameters known hyperparameters. Choosing the correct hyperparameter value is critical for optimal model performance. Currently, MLJ provides a satellite package called MLJ-Tuning for automate the hyperparameter tuning process. But the number of optimization algorithms available at MLJ-Tuning is limited. This goal of this proposal is to add Particle Swarm optimization as a new optimization algorithm.

Particle swarm optimization(PSO) is easy to understand optimization algorithm with highly active research community. In this proposal, I'll talk about the simple particle swarm optimization implementation that can handle single variable periodic function. I'll also talk about the possible PSO variants and show examples in Python, R and Julia. Finally I'll mention how PSO can be incorporated into MLJ.

2 Particle Swarm Optimization

Particle Swarm Optimization is a nature inspired numerical optimization method to minimize a function by generating and manipulating a swarm of particle(that represents possible solutions) and improve those solutions iteratively by simulating the behaviour of a bird flocking or a fish schooling.

This algorithm starts with a set of candidate solution that are distributed over the space that is assumed to contain the minimum of the objective function. These candidate solutions are known as particles.

At each step of the algorithm, the particle tries to find a more optimal position for themselves. The position update rules can be expressed as following,

$$x(t+1) = x(t) + v(t+1) \tag{1}$$

Where $x(t)$ and $x(t+1)$ are vectors that represents all particle position at time t and $t+1$ and $v(t+1)$ representing the velocity of the particle at time $t+1$

As particles move, it keeps track of its own best position. Each particles also communicate with other particles in the swarm to find best position. At each iteration particle finds its next position using a combination of its personal best from the past and current swarm's best particle position. This loop continues until a stop condition is met.

The velocity update rule for each particle has a stochastic and deterministic part. The deterministic parts are the following,

1. Inertia component: The velocity of the particle($v(t)$) from last iteration. This serves as a memory of previous movement.

2. Personal component: The distance between the particle's past best position and current position($x_b - x(t)$). This terms makes the particle want to move back to his past best position.

3. Social component: The distance between the swarm's best position and particle's current position($x_g - x(t)$). This terms makes the particle want to move towards the swarm's best position.

The personal and social component are multiplied by random values r_1 and r_2 that follows Uniform[0,1], and these terms introduces stochastic element to each particle movement. They are also multiplied by constant acceleration term c_1 and c_2.

So, the velocity update equation is the following,

$$v(t+1) = v(t)w + c_1 r_1(t)[x_b - x(t)] + c_2 r_2(t)[x_g - x(t)] \tag{2}$$

In each step of the algorithm, velocity is updated for each particle which is followed by position update. This process continues until a stopping condition is met. In my PSO implementation, I am ruining the algorithm for 25 iterations.

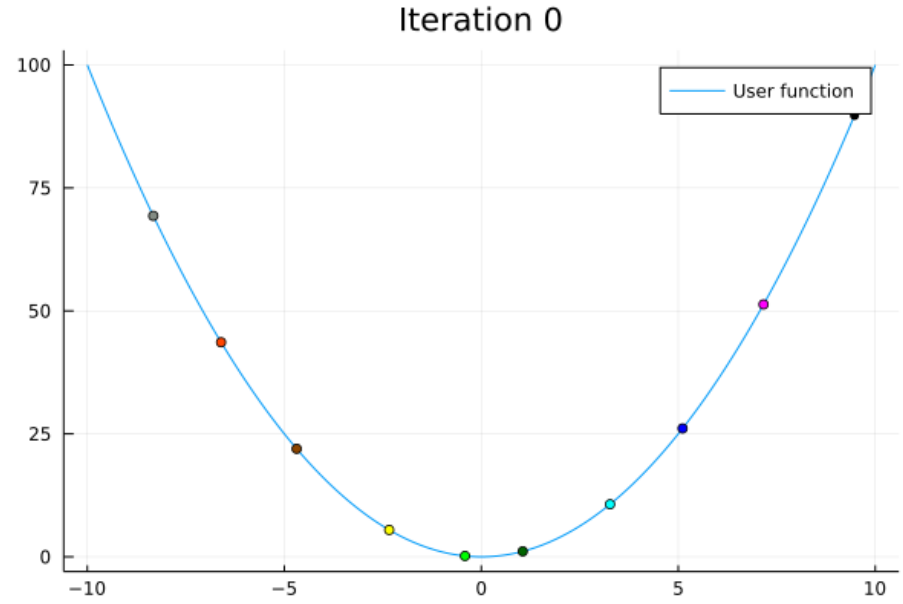
2.1 Vanilla PSO implementation

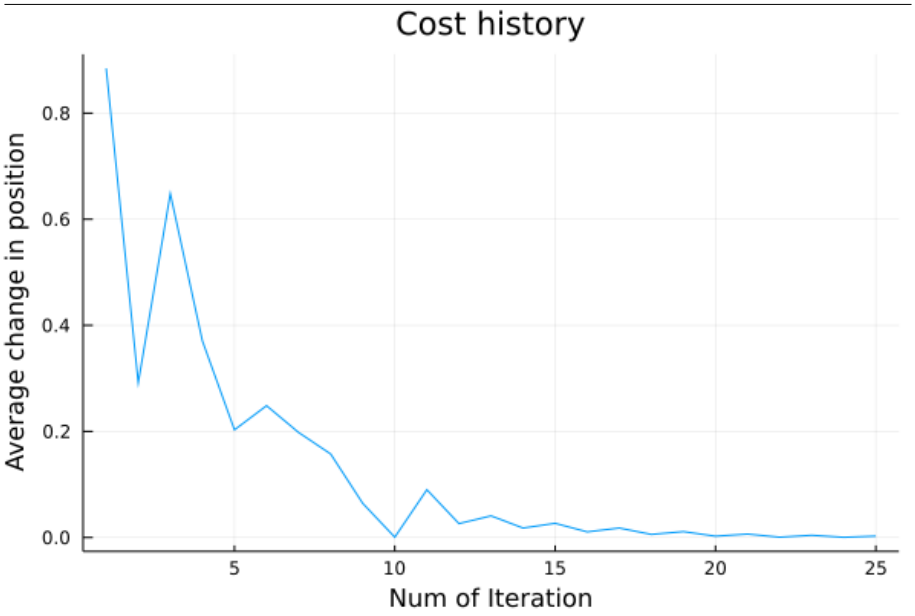
My vanilla PSO implementation can be found here, [Irf21a]. This implementation can minimizes a function of single variable. I'll demonstrate my PSO implementation by trying to optimize a single variable function, $f(x) = x^2$. Analytically we know that that the function minimizes at 0.

We will generate a swarm of 10 particles from -9 to 9 with interval 2. The graph of $f(x)$ is in blue and the swarm are represented as scatter point in unique colors(Iteration 0). The plot is an animation that will show the updated position of each particle on each iteration, from iteration 1 to 25. In the animated plot, we see a lot of oscillation in the first few iterations and the oscillation slowly dies down after each iteration. Note that the particles are starting to converge after 15 iterations.

This same observation is also evident in the cost history plot. On the x axis of the cost history graph we have the number of iteration and on y axis we have the cost which is defined as average change in position in two successive iterations. Note that there is a spike in the cost history at the beginning since the initial velocity of the particles are zero and they get too excited by the global minimum resulting in overshooting.

The values of particle position are also present in Table 1. As we can see that the particles are uniformly distribute at Iteration 0, and starts to converge in first few iterations, and by the 10th iteration they are already very close to our analytical solution. By 25th iteration all particles are very close to 0 - which is what we expect.


Iteration 0
User function
100
75
50
25
0
−10
−5
0
5
10


Cost history
Average change in position
0.8
0.6
0.4
0.2
0.0
5
10
15
20
25
Num of Iteration

Init(0)	1	2	...	5	...	10	...	25
-8.32	0.60	0.60	...	-0.25	...	-0.21	...	-0.0026
-6.60	0.38	0.38	...	-0.21	...	-0.16	...	-0.0021
-4.68	0.13	0.13	...	0.13	...	-0.08	...	-0.0012
-2.33	-0.16	-0.16	...	-0.13	...	-0.07	...	-0.0011
-0.41	-0.41	0.20	...	0.15	...	0.02	...	-0.0001
1.05	-0.60	-0.59	...	-0.28	...	-0.09	...	-0.0018
3.26	-0.88	-0.88	...	-0.50	...	-0.18	...	-0.0051
5.10	-1.12	-1.12	...	0.00	...	0.00	...	-0.0001
7.16	-1.38	-1.38	...	0.04	...	0.04	...	-0.0005
9.47	-1.68	-1.68	...	0.08	...	0.08	...	0.0023

2.2 Assumptions in vanilla PSO implementation

I've made the following assumptions in my implementation.

1. Particle initialization: 10 particles were generated uniformly in the range $[-9, 9]$, because we know the objective function is minimum at 0. It is important to make sure generated particles cover the area containing the optimal location otherwise it is difficult for PSO to converge. Particle velocity was set to zero. I also had tweak maximum velocity based on the problem.

2. Swarm connectivity: I am using global best - assuming all particles are connected with other particles in the swarm - similar to a complete graph. Other approaches could involved particles only connected to other particles closer to them based on some measure of distance such as Euclidean distance or spatial similarity.

3. Control parameters:

   Acceleration constant: c_1 and c_2 are positive acceleration constant that scales the contribution of personal and social component. It was set to 1.2.

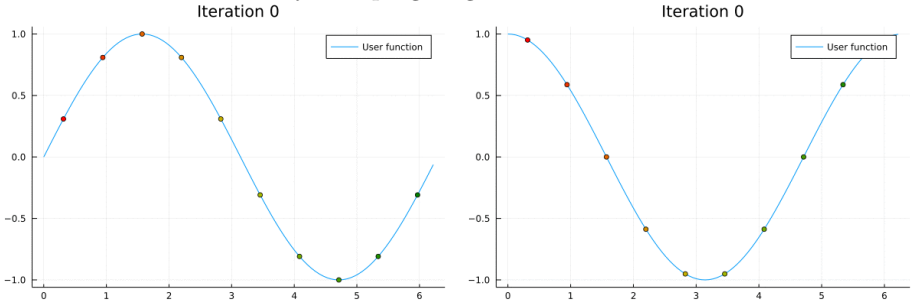
   Stochastic parameters: r_1 and r_2 are sampled from Uniform(0,1) in each iteration [Eng07]. These random values introduces the stochastic behaviour to the particles.

4. Stopping condition: There are many different stopping condition for PSO. I am chosing to stop the algorithm after 25th iteration as it's producing good result. The cost history also shows me that the average change is position is slowing down.

2.3 Adjusting for periodic function

As we know period functions are function that satisfied the $f(t+T) = f(t)$. I've updated my implementation that can find the minimum of a periodic function. I had to make a function that clamps the position of the particle between the

period. Here is the algorithm running on sin (left) and cos(right) function. I had to tweak the max velocity clamping to get stable result.



3 PSO Variants - Standard

The original paper published in 1995 by Kennedy and Eberhart had the following velocity and position update formula for each particle,

$$v(t+1) = v(t) + c_1 r_1 (x_b - x(t)) + c_2 r_2 (x_g - x(t))$$
$$x(t+1) = x(t) + v(t+1)$$

Where r_1, r_2 follows $Uniform(0, 1)$. The velocity is clamped between $[-V_{max}, V_{min}]$ to prevent the particles from flying out of the search space [KE95].

PSO is influenced by many parameters such as velocity clamping, particle inertia, dimension of the problem, number of particles, neighbour hood size(global vs local), number of iterations and many other. Some of these variants are described below with example usage in Python, R and Julia.

3.1 Inertia Weight

There is no precise method to chose the range of velocity therefore inertia weight(w) was introduced by [SE98] to replace velocity clamping which modified the velocity update equation to

$$v(t+1) = \boldsymbol{w} v(t) + c_1 r_1 (x_b - x(t)) + c_2 r_2 (x_g - x(t))$$

and it was also found that the algorithm converges if $w > \frac{1}{2}(c_1 + c_2) - 1$ [Van+07]. If w >1 then velocities increase over time, accelerating towards maximum velocity. If w <1 the particles decelerate until their reach minimum. Large values of w facilitate exploration and small w facilitate local exploration. Weight could also be dynamically by sampling from Gaussian distribution at each iteration and it could also linearly/non-linearly decreasing over time. Alfi and Fateh presented fuzzy PSO where particle dynamically adjust inertia weight according to particle best memories using non-linear model [AF11]. Inertia weight could not completely eliminate the need for velocity clamping.

3.1.1 Inertia Weight Example in R

MetaheuristicOpt package in R implements PSO with inertia weight variation. I've ran the optimization function on Rosenbrock ($a = 1, b = 100$) and the result of optimization is [0.9996966, 0.9993824] which very close to expected [1,1] [Irf21b] Section 1.

3.2 Velocity

3.2.1 Velocity clamping

As velocity increases in each iteration, it is important to clamp its value to prevent particles from flying out of search space. If V_{max} is too low swarm may be trapped in local minimum. If it's too high then particle may jump over good region. UsuallyV_{max} are selected to be a fraction of the search space on each dimension. That is,

$$V_{max} = \delta(x_{max} - x_{min})$$

The value of δ is problem dependent, and we can use cross-validation to find the best value [OES04]. A problem arises when all velocities are set to V_{min} or V_{max}. A solution to this problem is to change V_max over time linearly or exponentially.

3.2.2 Velocity constriction

Clerc proposed the use of constriction factor similar to inertia weight to balance between exploration and exploitation where velocities are constricted by a constant χ [Cle99],

$$\chi = \frac{2k}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|} \qquad \text{where } \phi = c_1 r_1 + c_2 r_2 \text{ and } k \in [0, 1]$$

This updates the velocity update equation to

$$v(t+1) = \chi[v(t) + c_1 r_1(x_b - x(t)) + c_2 r_2(x_g - x(t)))]$$

Here $\phi > 4$ guarantees convergence. It was developed as an alternate to velocity clamping. χ is evaluated to the range [0,1] which implies the velocity is reduced in each time step. The term k controls the exploration and exploitation ability of the swarm which is set to a constant value. Furthermore, Fan proposed a method where the velocity are multiplied by a scaling factor $(1 - \frac{t}{T})^h$ where t is the number of iteration, T is the maximum number of iteration and h is constant found by trial and error [Fan02].

3.3 Particle Connectivity

The performance of PSO strongly depends on how the particles are connected. In a fully connected swarm, each particle can learn about the best particle

position in the swarm. Particles are sometimes only connected to its neighbour and the locality of a particle can be calculated using Euclidean distance, spatial similarity, particle index or some other measure. This local variant do not converge as fast but has more exploitative features. Different social structures have been developed for PSO such as star, ring, wheel, pyramid, Von-Neumann and many more. In Von-Neumann social structure particles are connected in a grid, and it has been found to outperform other social networks in large number of problem. It was also found that in fully connected structures perform best for uni-modal problem and less connected structure perform better of multi-modal problem [Ken99]. Ni proposed to use random topology and analyzed its performance. [ND13]. Lim proposed PSO with that linearly increasing topology connectivity with time.

Suganthan proposed particle neighbour that grows with iteration [Sug99]. The search is initialized with a local best with number of particle in neighbour set to 2, and increases with each iteration until the the neighbour hood contains the entire swarm.

3.3.1 Local best Example in Python

PySwarm implements a local best PSO variant that uses ring topology and locality is defined in terms of distance computed using L2(Euclidean) norm. An example uses the implementation with neighbour size set to 3 and L2 distance set to 2 and trying to minimize the Rosenbrock function with $(a = 1, b = 1)$ and the result of the optimization is 0.99 and 0.99 which very close to what we expect. [Irf21c]

3.4 Particle Number and Initialization

Large number of particles allows larger parts of search space but increase computation time, and small number of particle lead to insufficient exploration and could result in local optima - therefore 20-100 was accepted according to Bratton [BK07]. It has also been showed that PSO has the ability to find optimal solution with small swarm size of 10 to 30 paticles [BE01].

Different initialization method has been proposed to ensure the search space is uniformly covered, such as Sobol sequence [PV+02] and non linear simplex method [PV02]. Gehlhaar suggest initializing particles in area that do not contain the minimum can be used to test the ability of the algorithm to find the solution.

3.5 Stopping Condition

PSO is an iterative algorithm. A number of termination criteria has been proposed,

1. Terminate when maximum number of iteration has been reached. Which is what I used in my Vanilla PSO implementation.

2. Terminate when an acceptable solution has been found. This requires prior knowledge of the optimal solution. Here we assume x^* is the optimum of the objective function, and the search process stop as soon as we find x which satisfies $f(x) \leq |f(x^*) - \epsilon|$. In this method the ϵ needs to be chosen with care.

3. Terminate when the particles are clustered up together. In this method we check if all particles are within a ϵ distance away from the global best particle. If all particles are within ϵ - we have a single cluster with all particles which terminates the algorithm. It is important to set ϵ with care to prevent premature terminate.

In some terminating condition, we are expecting the particles to clump up. But the particles could clump up in a local minimum, and we need to be aware of particle converging to local instead of global minimum.

3.6 Acceleration Coefficient

Acceleration coefficient c_1 and c_2 together with r_1 and r_2 control the stochastic influence of the personal and swarm influence on the velocity of a particle. c_1 and c_2 are also known as trust parameter. When $c_1 = c_2 = 0$ particle ignore personal and swarm best position keep flying a the current speed until flying out of the search space. If $c_1 > 0$ and $c_2 = 0$ all particles are independent hill climber. If $c_1 = 0$ and $c_2 > 0$ the entire swarm is attracted to a swarm's best particle.

In most algorithm c_1 and c_2 coexist in good balance, which means $c_1 \approx c_2$. If $c_1 >> c_2$ particles tends to favor their own personal best over swarm's best. Low values of c_1 and c_2 result in smooth particle movement and for high values particles make abrupt large movements.

Usually c_1 and c_2 are static. Suganthan reported that linearly decreasing c_1 and c_2 has no impact on performance [Sug99]. Ratnaweera proposed to decrease c_1 and increase c_2 linearly over time to facilitate early exploration and convergence at the end of the process. The update equations are the following,

$$c_1(t) = (c_{1,min} - c_{1,max})\frac{t}{n_t} + c_{1,max}$$
$$c_2(t) = (c_{2,max} - c_{2,min})\frac{t}{n_t} + c_{2,min}$$

Where $c_{1,max} = c_{2,max} = 2.5$ and $c_{1,min} = c_{2,min} = 0.5$

For an unconstrained simplified PSO system that included inertia, the trajectory of a particle surely converges if the following condition hold [Tre03],

$$1 > w > \frac{1}{2}(c_1 r_1 + c_2 r_2) - 1 \geq 0$$

The algorithm might still converge if the condition do not hold.

PSO with time varying acceleration coefficients are also proposed. Cai introduced time-varying accelerator coefficients which were adjusted according

to a predefined predicted velocity [CCT09]. Aminian introduced a fuzzy PSO method in which the inertia weight as well as acceleration coefficients were adjusted for each particle separately [AT13].

3.7 Cognition, Social and Selfless Model

In Cognition mode, the particles are only affected by their personal best. The velocity update equation is the following,

$$v(t+1) = v(t) + c_1 r_1 (x_p - x(t))$$

Cognition only model is slower in terms of number of iteration required to reach good solution, and it fails when velocity clamping and acceleration coefficient are small. Poor performance of this model was confirmed by [CD00].

The social only model ignores the personal best follows the swarm best,

$$v(t+1) = v(t) + c_2 r_2 (x_g - x(t))$$

It was found that social only model is faster and more efficient than full model in both static and dynamic environment. [Ken97; CD00]

Selfless model is similar to social model, but instead of using global best, local best is used. Kennedy showed that selfless model to be faster than social only model for few problem [Ken97].

4 PSO Variants - Advanced

All variants mentioned so far can be implemented by slightly modifying my vanilla PSO implementation. But so far the PSO variants we've described contain single objective, unconstrained and static. There are other variants that modified PSO extensively. For example, in some PSO variants both the velocity and position updates rules are substitute. In a variant both rules are substituted by a procedure that samples from a parametric probability density function such as a Gaussian distribution. Adaptive variant of this algorithm changes the standard deviation of the Gaussian distribution. The update rule of PSO can also be modified to a second order stochastic difference equation by Blackwell [Bla11]. But some researchers, instead of making the process complex, wanted to simplify standard PSO. For example, Guochu divided the swarm into three categories, better, ordinary and worst particles according to some fitness value [Che10].

In this section we will talk about PSO variants that are fused with other algorithms, contains multiple objective function and constrained.

4.1 Genetic Algorithm

Angelina provided first combined genetic algorithm with PSO. In this variant, particle are scored based on a fitness function given current position. The

best particles in the swarms replaces the position and velocity information of the bottom half [Ang98]. As expected this approach significantly reduces the diversity of the swarm because the selection pressure is too high [HI03].

4.2 Particle Reproduction

In another hybrid PSO variant particles can spawn offspring. Many reproduction schemes have been used with PSO. One of the first approach is Cheap-PSO by Clerc [Cle99] where a particle is allowed to generate a new particle, kill itself, or modify the inertia and acceleration coefficient. New particles are spawned if there is no significant improvement in the neighbourhood, or worst performing particle is killed of if there is sufficient improvement in the neighbourhood.

4.3 Particle Mutation

Gaussian mutation is another PSO variant where position of a particle is slightly adjusted after each iteration by adding a sample from Gaussian distribution. Miranda adjusted only the global best using the following,

$$x_g = x_g + \eta N(0, 1)$$

Here η is a learning parameter, which can be a static of dynamic based on some evolutionary strategy [MF02]. EP showed that Cauchy distribution results in better performance than Gaussian distribution because Cauchy distribution has thicker tails resulting in larger steps sizes - resulting in better exploration [YL96].

4.4 Cooperative Multi Swarm

In this PSO variant the swarm is divided into multiple groups where each groups performs different task or exhibits different behaviour. These tasks and behaviour can be dynamic as well. Individual particles can migrate between subswarms as well and multiple swarms can breed new particles.

Mohan divided all particles into two sub-swarms of equal size [AM02]. Each swarm can be either in attraction phrase, where particles are moving towards the global best ($c_1 = -1$ and $c_2 = 1$), or repulsion state, where particles are moving away from global best ($c_1 = 1$ and $c_2 = -1$). Swarm switch phase when a number of iteration is reached or particles do not show improvements. Cooperation between sub-swarms are achieved through selection of global best particles from each swarm.

4.5 Adaptive PSO

Adaptive PSO was proposed where algorithm parameter are tuned automatically using the evolutionary state which is combined with auxiliary search operators. Evolution state refers to the general behaviour of a particle throughout the algorithm. For example, at an early stage, the particles may be scattered

in various areas, and, hence, the population distribution is sparse. As the evolutionary process goes on particle would cluster up and converge to local or global minimum hence the population distribution would be different at this evolutionary stage.

These evolution states can be determined using cluster analysis. The 4 states particles can be in are exploration, exploitation, convergence and jumping-out which is represented by a variable f. If f is between 0 to 0.2 then the particle is said to be in convergence state, from 0.3 - 0.5 the particle is said to be in exploitation state, which is followed by 0.5 - 0.8 exploration state and 0.8 - 1 identifies as jumping out phase. The inertia weight is set a Sigmoid function of evolution state or f so the w will adapt to search environment characterized by f.

Besides dynamically setting the inertia coefficient, the acceleration coefficients are also dynamically adjusted. Both c_1 and c_2 start at 2 with the following strategies.

1. During exploration state c_1 is increasing and c_2 is decreasing, which means we are prioritizing personal best over swarm best.
2. In exploitation state c_1 is increasing slightly and c_2 is decreasing slightly.
3. In convergence state both c_1 and c_2 are increased slightly. It was found via experimentation that if we decrease c_1 at this stage, the swarm is very strongly attracted by the global best.
4. In Jumping out of state c_1 is decreasing and c_2 is increasing.

Also, the global best particle is perturbed by a sample from Gaussian distribution whose variance is linearly decreasing with iteration.

4.5.1 Adaptive PSO Example in Julia

Rosenbrock function with $a = 1$ and $b = 1$ is minimized using adaptive PSO which is available with Optim library written in Julia [Irf21d] Section 2. The result of the optimization is $a = 1$ and $b = 1$ which is what we expect.

4.6 Discrete PSO

2 years after the original publication Kennedy [KE97] published a discrete binary version of PSO where the space the particles will explore are discrete. In this variant we need to have particles in power of 2. Particles are defined as neighbour if the Hamming distance between the bit representation of their indices is one [AA03]. In this modification the velocity update equation remains the same, except the position of particles are represented by 0s and 1s, and the velocity is clamped between [0, 1] which is achieved via logistic transformation $S(v_t) = \frac{1}{1-e^{v_t}}$. The position update expression was updated,

$$x(t+1) = \begin{cases} 1 & Uniform(0,1) < S(v_{t+1}) \\ 0 & otherwise \end{cases} \tag{3}$$

A popular particle swarm optimization Python library is PySwarm that implements discrete PSO.[Les17]

4.7 Multi-objectives PSO

By dynamically updating particle neighbourhood and calculating local best particle position, PSO can be used to address problem that involves multiple objective function. We are interested in generating the Pareto front, which is a set of Pareto optimal solution where it is not possible to improve one objective function without worsening at least one other objective function.

In this approach only one objective is optimized at a time. The algorithm used to search for local optima in each generation is defined as follows:

1. Calculate the distances of the current particle from other particles in the fitness value space of the first objective function (not the variable space).
2. Find the nearest m particles as the neighbors of the current particle based on the distances calculated above (m the neighborhood size).
3. Find the local optima among the neighbors in terms of the fitness value of the second objective function.

4.7.1 Multi-objective PSO Example in R

MOPSO-CD is a variant of multi-objective PSO that uses crowding distance(CD) computation to ensure an even spread of non-dominated solutions. Here in an example in R optimizing Viennet function[Irf21b] Section 2.

4.8 Constrains

Particles might move to region that is not feasible, and PSO variants were developed to handle situations like that. Some very simple approaches are,

1. If the number of infeasible particles are small, not allowing those particle to be selected as personal or global best could result in them being pulled back in the feasible space.
2. Reject infeasible particles and replace them with newly generated ones.
3. Add penalty for infeasible particles [PV+02].
4. Apply repair operation on the feasible particles to move them back to feasible space. Hu and Eberhart developed an approach where particles are not allowed to be attracted by infeasible particles. [HE02]

4.8.1 Constrain Example in R

An example in written in R that is running PSO with inertia weight on 1 objective function and 4 constrains. Each time the particle position do no satisfy constrain, a penalty value (1000) is added to it [Irf21b] Section 3.

5 PSO Standardization Effort

As we've seen there is no correct way to implement PSO. To streamline the PSO literature a standardization body has developed Standard Particle Swarm Optimisation standard(SPSO). Since 2006 there has been 3 successive standards namely SPSO 2006, 2007 and 2011. PSO formula for each standard is slightly different because they took advantage of latest theoretical analysis as that time [Cle12].

6 Existing Implementations of PSO

A gold standard of any algorithm is the amount of people using it. As more people use the same implementation, the bugs are ironed out over time making the implementation more reliable. Therefore it is important to find an existing implementation in an existing library before re-implementing it in a new language. In this section I'll explore existing implementation of Particle Swarm algorithm that's part of a larger library.

6.1 C++

1. **OptimLib** is a library for optimizing non-linear function. This library has two PSO variants, PSO with dynamic inertia weight and and PSO with differentially-perturbed Velocity. I could not locate the source of the algorithm [OHa].

2. **PaGMO** is scientific library for massively parallel optimization and it contains PSO variant with non dominant sorting and self-adaptive differential evolution(NSPSO). Source paper of the algorithm is available on their website[BI20]

6.2 Python

Despite being the most popular scientific library Sciki-optimize and Scipy do not have PSO implemented in their optimization module [com20]. But I was able to find other library that is either dedicate for swarm algorithms or include PSO. We can use PySwarm with SKLearn to tune hyperparameter for machine learning models.

1. **Pymoo** is a multi objective optimization library that implements vanilla PSO with global best perturbation [BD20].

2. **PyGMO** is another multi objective optimization library that is a python wrapper for the aforementioned C++ library PaGMO.

3. **PySwarm** is an extensible research toolkit for particle swarm optimization library. It supports continuous and discrete search space. An example using PySwarm is available in section 3.3.1

6.3 R

1. **Automl** package fits from simple regression to deep neural networks either with gradient descent or metaheuristic, using automatic hyper parameters tuning and custom cost function [Bou20a]. Their official documentation also includes content on hyperparameter tuning using PSO [Bou20b].

2. **metaheuristicOpt** package implements PSO with inertia weight variant [Sep+19]. An example is implemented in section 3.1.1 in this proposal.

6.4 Julia

1. **Optim.jl** implement PSO variant with Adaptive PSO. The algorithm is briefly explained in section 4.7 in this proposal with an example on section 4.7.1.

2. **Manopt.jl** is an optimization on Manifolds in Julia. This PSO variant includes a gradient component. The operations involved in the PSO algorithm are redefined using concepts of differential geometry [BIA10].

7 Machine Learning for Julia(MLJ) Library

MLJ is a toolbox that aggregates implementations of machine learning algorithm and provides a common and consistent interface for the user. Julia type system allows MLJ to introduced new features such as model pipelining which allows ease in multi model setup. This library also provides scientific type which allows specific interpretation of the data instead of relying on binary encoding. There are 4 scalar scientific type: OrderedFactor and Multiclass for categorical data and Continuous and Count for non-categorical data. These scientific types makes it easy to find an appropriate model for a given data-set.

An example of MLJ is shows by training a logistic model to predict outcome of a house after it was diagnosed and treated. This example demonstrates the most basic structure of MLJ [Irf21e]. We start by setting the appropriate scientific type of the data(line 7 - 12), and then we instantiate Logistic regression model using the default parameters. Each model is essentially a named tuple and we can query input scitype and target scitype to find the type of data the model is expecting. After model initialization and one of the hyperparameter was changed to 100 in line 23.

Once we have a model instance and our data ready, we bind them together into a machine. The machine is responsible for saving the trained model information. For example, in case of linear regression the machine would hold the values of regression coefficient. This step is followed by training the model in line 26 using the fit function. Once we have a trained model we can use our test data to predict the accuracy of the model in line 27. We compared the prediction with the original data set to find the accuracy of the model in line 28.

7.1 Hyperparameter Tuning

A given model could perform better we tune it's hyperparameters. We can use the LearningCurve function to test the model performance over a range of hyperparameters against some measure(i.e. cross entropy, root mean square, brier score etc). To tune multiple hyperparameters for a composite model, MLJ provides TunedModel struct which accepts a range of hyperparameter, a measure and tuning strategy. Currently the following tuning strategies are available,

1. search models generated by an arbitrary iterator
2. grid search
3. Latin hypercubes
4. random search
5. structured tree Parzen estimators

The goal of this proposal is to implement a new tuning strategy called Particle Swarm Optimization.

7.2 Implementing new tuning Strategy

To implement a new tuning strategy we need add a new algorithm to MLJ Tuning src/strategies or by implementing MLJTuning's Tuning Strategy Interface with defaults. Tuning in MLJ is an iterative process. At each iteration performance of a model is evaluated. When all iteration of the algorithm are complete, the optimal model is selected by some selection heuristic to a history generated according to a specific tuning strategy. The fields of a tuning strategy are called tuning hyperparameters and they control the tuning strategy. Selection heuristic is a rule which decides on the "best model" given the model evaluations in the tuning history [Tur].

8 Pre-GSoC and Deliverable

The aim of the project is to investigate and implement PSO and it's variants to MLJ library and document them thoroughly. So far I've implemented a simple PSO and extensively studies many PSO variants published in the last 25 years. I've also familiarized myself with MLJ. At the beginning of Summer, I will start by implementing a vanilla single variable PSO and test it with simple models(i.e. Simple/Multiple Linear regression). From there we will take it in different directions. For example, we will include dynamic weight and acceleration coefficient variant. We will also add setting that allows the use to change the particle connectivity and life cycle. Beyond single object optimization, we will also look into multi object PSO, multi constrain PSO and both multi objective and multi constrain PSO. For example will will implement non dominated

sorting PSO variant for multi object optimization since this algorithm is also implemented in other language libraries.

I'll also document the code thoroughly. I'll run my implementations on well known test functions to test for the accuracy of my implementation. Currently MLJ has 162 model in it's repository and PSO has numerous variants. My goal is to implement as many PSO algorithms as possible to cover a wide range of PSO variants.

9 Biographical Information

My name is Quazi Irfan, and I am a graduate student studying at South Dakota State University. I am graduating this Summer 2021. I did my undergraduate in Computer Science and have extensive programming experience. I mostly programmed in C++, Java, Python and R. I've recently moved to Julia because a high level language with low level performance is very appealing to me. So far, I have studied Algorithm analysis and Design, Programming Language, Compiler Design, Statistical Inference, Regression, Multivariate Statistics and Linear Algebra to name a few. Currently I am working on my thesis on robot localization using inertial navigation system where I am extensively studying signal processing. I have briefly studies meta-heuristics algorithm in my Algorithms course. I have not taken any course named Machine Learning, but I am familier with ML topic via osmosis. Because I've had course that went over Linear and Polynomial regression, Classification(LDA, QDA, KNN), Resampling method(Cross validation, Bootstrap), Linear model selection(Ridge, LASSO, PCR), Tree based methods, SVM and unsupervised learning.

Beyond course work, I've implemented a working compiler for a subset of Ada language [Qua21], and lead the Robotics club software team where we've implemented a path finding algorithm for the robot [Irf]. In the past, I have worked as a research assistant where I had to debug a very old code base written in Java. I also did undergraduate research on building haptic feedback gloves for virtual reality [Irf+18]. I've also contributed to open source project(jMonkeyGame Engine) in the past, and strongly familiar with Git and Github workflow.

I am a highly motivated self learner, and I am confident that I'll be able to solve a problem within a reasonable time given I have proper background knowledge and reading the right resource.

Note: I am taking only one research credit this summer, so I'll have wide availability to work on GSoC project. I am planning to spend 20 hours per week or more if needed. I will always available through slack/email/phone.

References

[AA03] Ashraf M Abdelbar and S Abdelshahid. "Swarm optimization with instinct-driven particles". In: *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.* Vol. 2. IEEE. 2003, pp. 777–782.

[AF11] Alireza Alfi and Mohammad-Mehdi Fateh. "Intelligent identification and control using improved fuzzy particle swarm optimization". In: *Expert Systems with Applications* 38.10 (2011), pp. 12312–12317.

[AM02] Buthainah Al-Kazemi and Chilukuri K Mohan. "Multi-phase generalization of the particle swarm optimization algorithm". In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600).* Vol. 1. IEEE. 2002, pp. 489–494.

[Ang98] Peter J Angeline. "Using selection to improve particle swarm optimization". In: *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98TH8360).* IEEE. 1998, pp. 84–89.

[AT13] Ehsan Aminian and Mohammad Teshnehlab. "A novel fuzzy particle swarm optimization". In: *2013 13th Iranian Conference on Fuzzy Systems (IFSC).* IEEE. 2013, pp. 1–6.

[BD20] J. Blank and K. Deb. "Pymoo: Multi-Objective Optimization in Python". In: *IEEE Access* 8 (2020), pp. 89497–89509.

[BE01] F van den Bergh and Andries P Engelbrecht. "Effects of swarm size on cooperative particle swarm optimisers". In: *Proceedings of the 3rd annual conference on genetic and evolutionary computation.* 2001, pp. 892–899.

[BI20] Francesco Biscani and Dario Izzo. "A parallel global multiobjective framework for optimization: pagmo". In: *Journal of Open Source Software* 5.53 (2020), p. 2338. DOI: 10.21105/joss.02338. URL: https://doi.org/10.21105/joss.02338.

[BIA10] Pierre B. Borckmans, Mariya Ishteva, and Pierre-Antoine Absil. "A Modified Particle Swarm Optimization Algorithm for the Best Low Multilinear Rank Approximation of Higher-Order Tensors". In: *Swarm Intelligence.* Ed. by Marco Dorigo et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 13–23.

[BK07] Daniel Bratton and James Kennedy. "Defining a standard for particle swarm optimization". In: *2007 IEEE swarm intelligence symposium.* IEEE. 2007, pp. 120–127.

[Bla11] Tim Blackwell. "A study of collapse in bare bones particle swarm optimization". In: *IEEE Transactions on Evolutionary Computation* 16.3 (2011), pp. 354–372.

[Bou20a] Alex Boulangé. *automl: Deep Learning with Metaheuristic.* R package version 1.3.2. 2020. URL: https://CRAN.R-project.org/package=automl.

[Bou20b] Alex Boulangé. *automl: PSO hyperparameter tuning*. R package version 1.3.2. 2020. URL: https://aboulaboul.github.io/automl/articles/automl.html#x1-80000.2.5.

[CCT09] Xingjuan Cai, Yan Cui, and Ying Tan. "Predicted modified PSO with time-varying accelerator coefficients". In: *International Journal of Bio-Inspired Computation* 1.1-2 (2009), pp. 50–60.

[CD00] Anthony Carlisle and Gerry Dozier. "Adapting particle swarm optimization to dynamic environments". In: *International conference on artificial intelligence*. Vol. 1. Citeseer. 2000, pp. 429–434.

[Che10] Guochu Chen. "Simplified particle swarm optimization algorithm based on particles classification". In: *2010 Sixth International Conference on Natural Computation*. Vol. 5. IEEE. 2010, pp. 2701–2705.

[Cle12] Maurice Clerc. "Standard Particle Swarm Optimisation". 15 pages. Sept. 2012. URL: https://hal.archives-ouvertes.fr/hal-00764996.

[Cle99] Maurice Clerc. "The swarm and the queen: towards a deterministic and adaptive particle swarm optimization". In: *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*. Vol. 3. IEEE. 1999, pp. 1951–1957.

[com20] Scipy community. *Optimization scipy optimize*. R package version 1.3.2. 2020. URL: https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html.

[Eng07] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[Fan02] Huiyuan Fan. "A modification to particle swarm optimization algorithm". In: *Engineering Computations* (2002).

[HE02] Xiaohui Hu and Russell Eberhart. "Solving constrained nonlinear optimization problems with particle swarm optimization". In: *Proceedings of the sixth world multiconference on systemics, cybernetics and informatics*. Vol. 5. Citeseer. 2002, pp. 203–206.

[HI03] Natsuki Higashi and Hitoshi Iba. "Particle swarm optimization with Gaussian mutation". In: *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706)*. IEEE. 2003, pp. 72–79.

[Irf] Quazi Irfan. *quazi-irfan/RoverMapping*. URL: https://github.com/quazi-irfan/RoverMapping.

[Irf+18] Quazi Irfan et al. "Building an exoskeleton glove on virtual reality platform". In: *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE. 2018, pp. 0645–0650.

[Irf21a] Quazi Irfan. *quazi-irfan/Julia$_P$SO$_G$SoC21*. 2021. URL: https://www.github.com/quazi-irfan/Julia_PSO_GSoC21/blob/master/vanillaPSO.jl.

[Irf21b] Quazi Irfan. *quazi-irfan/Julia$_P$SO$_G$SoC21*. 2021. URL: https://github.com/quazi-irfan/Julia_PSO_GSoC21/blob/master/inertiaPSO.r.

[Irf21c] Quazi Irfan. *quazi-irfan/Julia$_P$SO$_G$SoC21*. 2021. URL: https://github.com/quazi-irfan/Julia_PSO_GSoC21/blob/master/variantPSO.py.

[Irf21d] Quazi Irfan. *quazi-irfan/Julia$_P$SO$_G$SoC21*. 2021. URL: https://www.github.com/quazi-irfan/Julia_PSO_GSoC21/blob/master/optimPSO.jl.

[Irf21e] Quazi Irfan. *quazi-irfan/Julia$_P$SO$_G$SoC21*. 2021. URL: https://www.github.com/quazi-irfan/Julia_PSO_GSoC21/blob/master/MLJ.jl.

[KE95] James Kennedy and Russell Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95-international conference on neural networks*. Vol. 4. IEEE. 1995, pp. 1942–1948.

[KE97] James Kennedy and Russell C Eberhart. "A discrete binary version of the particle swarm algorithm". In: *1997 IEEE International conference on systems, man, and cybernetics. Computational cybernetics and simulation*. Vol. 5. IEEE. 1997, pp. 4104–4108.

[Ken97] James Kennedy. "The particle swarm: social adaptation of knowledge". In: *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*. IEEE. 1997, pp. 303–308.

[Ken99] James Kennedy. "Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance". In: *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*. Vol. 3. IEEE. 1999, pp. 1931–1938.

[Les17] James V. Lester. *pyswarms.discrete package*. 2017. URL: https://pyswarms.readthedocs.io/en/latest/api/pyswarms.discrete.html.

[MF02] Vladimiro Miranda and Nuno Fonseca. "EPSO-best-of-two-worlds meta-heuristic applied to power system problems". In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02Th8600)*. Vol. 2. IEEE. 2002, pp. 1080–1085.

[ND13] Qingjian Ni and Jianming Deng. "A new logistic dynamic particle swarm optimization algorithm based on random topology". In: *The Scientific World Journal* 2013 (2013).

[OES04] Mahamed G Omran, Andries P Engelbrecht, and Ayed Salman. "Image classification using particle swarm optimization". In: *Recent advances in simulated evolution and learning*. World Scientific, 2004, pp. 347–365.

[OHa] Keith O'Hara. URL: https://www.kthohr.com/optimlib.html.

[PV+02] Konstantinos E Parsopoulos, Michael N Vrahatis, et al. "Particle swarm optimization method for constrained optimization problems". In: *Intelligent Technologies–Theory and Application: New Trends in Intelligent Technologies* 76.1 (2002), pp. 214–220.

[PV02] KE Parsopoulos and MN Vrahatis. "Initializing the particle swarm optimizer using the nonlinear simplex method". In: *Advances in intelligent systems, fuzzy systems, evolutionary computation* 216 (2002), pp. 1–6.

[Qua21] Quazi. *quazi-irfan/Mini-Ada-Compiler*. 2021. URL: `https://github.com/quazi-irfan/Mini-Ada-Compiler`.

[SE98] Yuhui Shi and Russell Eberhart. "A modified particle swarm optimizer". In: *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*. IEEE. 1998, pp. 69–73.

[Sep+19] Lala Septem Riza et al. *metaheuristicOpt: Metaheuristic for Optimization*. R package version 2.0.0. 2019. URL: `https://CRAN.R-project.org/package=metaheuristicOpt`.

[Sug99] Ponnuthurai N Suganthan. "Particle swarm optimiser with neighbourhood operator". In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. Vol. 3. IEEE. 1999, pp. 1958–1962.

[Tre03] Ioan Cristian Trelea. "The particle swarm optimization algorithm: convergence analysis and parameter selection". In: *Information processing letters* 85.6 (2003), pp. 317–325.

[Tur] Alan Turing. *alan-turing-institute/MLJTuning.jl*. URL: `https://github.com/alan-turing-institute/MLJTuning.jl`.

[Van+07] Frans Van Den Bergh et al. "An analysis of particle swarm optimizers". PhD thesis. University of Pretoria, 2007.

[YL96] Xin Yao and Yong Liu. "Fast Evolutionary Programming." In: *Evolutionary programming* 3 (1996), pp. 451–460.