Deep Swarm: Nested Particle Swarm Optimization

Russell C. Eberhart
Consultant

Indianapolis, IN, USA
rceberhart@gmail.com

Doyle J. Groves
Data Scientist
Proofpoint, Inc.

Indianapolis, IN, USA
dgroves@proofpoint.com

Joshua K. Woodward
Electrical and Computer Engineering Department
Indiana University Purdue University Indianapolis
Indianapolis, IN, USA
jkwoodwa@umail.iu.edu

Abstract— A new generation of particle swarm optimization (PSO) has been developed that automatically evolves optimal or near-optimal values for parameters of the PSO algorithm such as population size and neighborhood size, and, if used, parameters of associated neural network(s), such as number of hidden processing elements (PEs). Called Deep Swarm, it is a nested version of PSO, and comprises swarms within a swarm.

Keywords—particle, swarm, optimization, deep, nested, adaptation

I. Introduction

In the canonical version of particle swarm optimization (PSO), a system is initialized with a random set of solutions. Each potential solution is also assigned a random velocity, and the potential solutions, called *particles*, are then "flown" through the problem space.

In the problem space, each particle keeps track of its coordinates that are associated with the best solution (fitness) it has achieved so far. This value is called "pbest."

The other "best" value tracked by the global version of the particle swarm optimizer is the overall best value gbest, and its location, obtained so far by any particle in the population. There is also a local version of PSO in which, in addition to pbest, each particle keeps track of the best location, called "lbest," attained within a local topological neighborhood of particles.

At each time step, each particle is stochastically accelerated toward its pbest and gbest locations (in the global version), or toward pbest and lbest (in the local version.)

This original canonical PSO, developed by Jim Kennedy and Russ Eberhart, was first reported in 1995 [1,2].

The maximum velocity V_{max} was difficult to deal with until the concept of an inertia weight was incorporated into PSO [3,4]. A common choice for the inertia weight is (0.5 + *rand*/2) which balances exploration and exploitation, especially important in a dynamic environment. The maximum velocity is then set to the dynamic range on each dimension (for each element) of the particle [5]. Reference [5] includes a more detailed discussion of PSO, including equations and additional resources.

A significant portion of the time and effort when implementing PSO is spent experimenting with configuring the application for the algorithm, and configuring the algorithm for the application. Often, an application involves a neural network, the weights of which are adapted/optimized with PSO. Other neural network parameters such as the number of processing elements (PEs), and the slopes of PE transfer functions are also sometimes optimized.

In the past decade, several researchers have studied ways to optimize PSO configuration. Results from these attempts, often labeled as versions of self-adaptive PSO (SAPSO), have been mixed.

Reference [6] comprises an evaluation of eight SAPSO approaches. This study utilizes a benchmark function specially-formulated to avoid stagnation. Earlier studies, such as that reported in [7], examine SAPSO behavior on a variety of benchmark functions. In each of these functions, the optimum and feasible space are predefined.

Only a few PSO parameters are considered in each study. Typically, the inertia weight, and the social and cognitive control parameters c_1 and c_2 are tuned.

In practical engineering applications such as that described in this paper, it is desired to optimize a significantly larger number of system parameters. Our applications often involve optimizing a neural network application for classification, or a complex scheduling application.

In addition to the three parameters mentioned above, we want to optimize parameters, such as the number of particles in a swarm and the number of hidden PEs in a neural network. In the work reported here for the Perl implementation, up to eleven parameters are optimized simultaneously.

This is the author's manuscript of the article published in final edited form as:

Eberhart, R. C., Groves, D. J., & Woodward, J. K. (2017). Deep swarm: Nested particle swarm optimization. In 2017 IEEE Symposium Series on Computational Intelligence (SSCI) (pp. 1–6). https://doi.org/10.1109/SSCI.2017.8280920

This paper presents a new generation of particle swarm optimization that has been developed to automatically evolve optimal or near-optimal values for parameters of the PSO algorithm, such as population size and velocity factor, and, if used, parameters of associated neural network(s), such as number of hidden PEs. Called Deep Swarm, a nested version of PSO, it comprises swarms within a swarm.

Initially developed for a cybersecurity application, the analysis of large volumes of malware events, Deep Swarm significantly reduces the time and effort to develop a complex application that utilizes PSO. It allows the optimization of a system tailored to each application, thus going significantly beyond tuning a limited number of parameters based on benchmark functions and mathematical analyses.

II. Configuration and Process

A. Configuration

The current configuration of Deep Swarm comprises multiple swarms within a swarm. Its initial application is for adapting/optimizing a neural network to classify malware events. Daily volume in the cybersecurity application is approximately 200,000 malware-related entities with reputations (usually domains or IP addresses, but can be other attributes) per day [8].

Challenges associated with the application included:

- Dealing with the constantly changing environment
- Presenting information on malware attacks that is practical and useful
- Making information actionable for humans and automating some actions
- Emphasizing accuracy and simplicity

The first attempt at a classification tool was to evolve a traditional feedforward neural network with canonical PSO. In order to achieve acceptable system performance, it was found that a significant amount of time was required to select swarm and neural network parameters, such as the number of particles in the swarm, the number of hidden processing elements in the neural network, etc.

This led the development team to devise a tool that would self-optimize, and thus be useful in dynamic environments. The tool became Deep Swarm.

Discussions with the users resulted in the choice of a three-category classification system. Events are classified as Red (bad; requires urgent attention), Yellow (neutral; keep an eye on this), or Green (ok; can be ignored for now). This is significantly simpler than presenting multiple reputation categories, each with a severity, for each event. It is also visual, and easier to understand. An accuracy of 90 percent correct was specified for the initial proof-of-concept system.

In our initial application, the outer (primary, or shell) swarm uses a gbest particle swarm with five elements (dimensions), and one user-specified parameter for the population size. Maximum velocities are specified by the dynamic range on each dimension, as is now standard in PSO.

The swarm's elements are velocity factor, neighborhood size, number of particles (population size), number of hidden PEs in the neural network, and upper/lower limits of weights in the neural network. These are being optimized/adapted along with classification neural network weights that are optimized/adapted in the inner swarms.

Note that velocity factor is not included in most PSO implementations, but it illustrates the fact that non-standard parameters can be experimented with and retained or discarded. Velocity factor was not implemented as one of the 11 parameters that can be optimized in the Perl Script Deep Swarm implementation.

These five elements in the outer swarm are initialized and run within the ranges in Table I, which were selected by experience and with assistance of analysis tools (see *Process Commentary Section* below).

TABLE I: PARAMETER INITIALIZATION RANGES

Parameter	Initialization range	Limit during run
Population size	15-50	10-100 (integer)
Neighborhood size	1-5	1-10 (integer)
Velocity factor	.5-1.5	0.2-2.0
Weight limits	+/- 2.0-12.0	+/- 0.5-20
No. of hidden PEs	5-15	3-20

B. Process

For each of the population members (particles) of the outer (primary, shell) population, another swarm is configured using the five values for an inner (network weights) swarm. In our initial application, inputs to the inner swarm are severities of malware event reputation categories.

There are 41 categories in each malware-related entity, but we initially used only 18, since they were the only ones with non-zero values in our initial pattern sets. Examples of categories are Command-and-Control, and IP-Check.

Each reputation category has been assigned a severity (0-127) and a characterization (bad, neutral, OK). Each entity (event) can trigger multiple categories, each with its unique severity.

Each inner swarm is run for the fixed number of iterations specified in an input file, the *parameters.run* file, and the best severity value (lowest weighted sum of mistakes based on the confusion matrix of Table II) is used to select values of gbest for the adaptation of the five-element outer (primary) swarm.

TABLE II: CONFUSION MATRIX

Column as row error	Red	Yellow	Green
Red	0.00	0.25	1.00
Yellow	0.50	0.00	0.25
Green	1.00	0.25	0.00

In each shell swarm iteration, each inner swarm's elements representing network weights are initialized randomly within the range of weight limits defined for that particle by the associated outer (primary, shell) swarm. The patterns are then used as inputs to evolve the weights for a neural network for the number of iterations specified in *parameters.run* for each particle.

Values of weights, best error (severity), and the associated five swarm parameters for the gbest particle are stored. (The five parameter values remain fixed during the inner swarms' runs.) Note that the file *parameters.run* file is written to after each iteration, as are the TEMP files.

The outer swarm is then flown one iteration, using the fitness (severity) values for the inner swarms to select gbest. Now there are new values for the five parameters for each inner particle obtained from flying the outer swarm, and the process is repeated for the specified number of generations.

Our Deep Swarm/nested swarm adaptation can be used for other parameters, such as optimizing the slope *k* of the individual processing element sigmoid transfer function, optimizing the type of transfer function, etc. Each additional parameter being optimized simply adds a dimension to the outer (shell) swarm described above.

We have initially obtained good results by running the shell swarm significantly fewer iterations than the inner (network weights) swarms. Typical numbers of iterations for the outer and inner swarms for our malware classification application are 20 +/- 10 and 200 +/- 100, respectively. Note that we have very little experience so far with this aspect of Deep Swarm adaptation.

C. *Process Commentary*

Every data set intended for a classification application has its natural level of "groupability" – how cleanly separable will classes be in the input dimensional space? What is the optimal number of "natural" states of this input data?

Getting at this number of natural states was inspired by earlier work in topic detection from unknown text collections – how many topics optimally describe what is in the complete collection? Reference [9] describes word-adjacency-graph (WAG) modeling as a means to model all related words, then partition the model into clusters that represent content topics.

For our 18-dimensional numerical input data, we used a two-step process in the exploratory phase to look for an idea of the optimal number of states. This provided insight to the number of hidden processing elements needed in the neural network classifier, guiding the choice of the initialization range for this parameter.

We first performed a principal components analysis and found the first three components together describe 67 percent of the variance, sufficient to loosely characterize the big groups found. Charting these in three dimensions shows clear grouping for about ten clusters, if not a few more, as seen in Fig. 1, from Tensorflow's Embedding Projector tool [10].

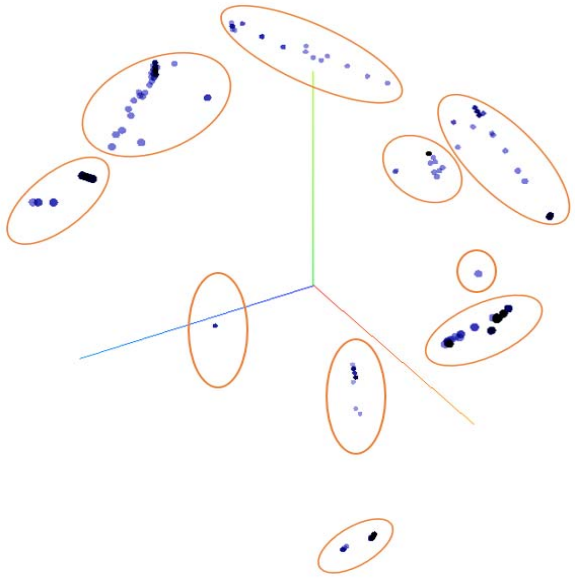

Fig. 1. 3D view of first three principal components.

Results of K-means with ten groups are shown in Fig. 2, with good definition on each of nine groups, and group 10 as an outlier catch-all.

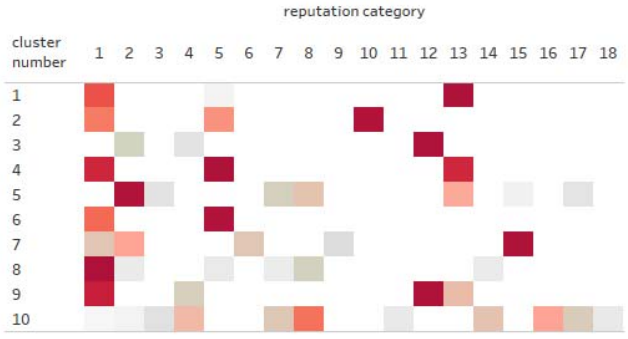

Fig. 2. Mapping 18 inputs and ten clusters.

For example, Group 1 shows strong signal in reputation category 1 and 13 only. Group 2, rep categories 1, 5 and 10. Whereas most groups are defined by signals in only a few categories, Group 10 (the "others") has at least some signal in eleven categories.

Knowing that ten basic types of input are to be dealt with here, ultimately toward successful classification, helps inform the design of the neural network topology, particularly the number of hidden layer PE's if not the number of hidden layers as well.

III. Malware Event Data Classification and Analysis

Deep Swarm was used to develop and test a classifier for a sample 548-pattern malware event database. The initial code for Deep Swarm was written in C#. Other versions including one in Perl, discussed later, are being implemented.

This training data set was developed by interviewing subject matter experts who were shown scenarios of malware-related entities and relevant attributes like reputation scores in various categories. Given those facts, they assigned a threat level category of Red, Yellow, or Green to each example.

The following discussion refers to ten runs made of the Deep Swarm system. The outer swarm parameters varied as follows: number of generations was either five or ten; number of particles, which is the number of inner swarms, varied from ten to 40. Within the inner swarms, the number of iterations varied from 200 to 500.

Over all ten runs, there was a total of 520 errors (misclassifications), or an average of 52 per run, with 49-54 errors for individual runs. Details appear in Table III.

Of the 50 problem patterns, 17 (34 percent) were determined to have been misclassified originally. Incorporating these 17 revised classifications into the data set results in a reduction of errors, and a reduction of severity. Data runs are still being made as of the writing of this paper, but preliminary results indicate an increase from 90.5 to approximately 92.5 percent correct, and a decrease in average severity to around 16.0.

TABLE III. Results from 10 Runs of Deep Swarm

Errors	Severity	Pop. size	Neigh.	Wt. limit	Hid. PEs
52	17.25	24	4	7.7	6
52	17.25	20	5	7.3	8
49	16.75	66	4	8.4	7
54	18.25	26	2	4.4	15
53	17.5	37	5	7.9	10
53	17.5	38	5	5.3	11
53	17.5	37	6	3.8	9
51	17	40	3	2.2	7
51	17	39	6	10.5	17
52	17.25	32	5	13.6	12

The distribution of errors was unexpected. Fifty patterns accounted for 490 of the errors, or an average of 49 errors per run. Each of these patterns was misclassified in at least eight runs. Only 18 patterns accounted for the remaining 30 errors, and they each accounted for three or less errors, except for one pattern with four.

The 50 patterns with at least eight errors each were obvious candidates for review. The system could seldom be evolved to classify them 'correctly.'

These patterns were provided for reclassification to the expert who originally classified them. They were not told how they were previously classified, and were discouraged from referring to their original classification.

A second expert has also reclassified the 50 patterns. The number of different reclassifications is similar to that of the first expert, but the overlap between the two experts is only about two-thirds.

With the participation of both experts, the project team is currently determining how best to incorporate differences in human inputs into the Deep Swarm tool.

IV. Implementation

In application areas for which the environment is relatively stable, once Deep Swarm is used to configure and optimize the classifier such as a neural network, the classifier's training can be "touched up" by re-training it using recently acquired data patterns. This requires relatively little computing time.

However, a re-optimization of the entire Deep Swarm system, which requires one or two orders of magnitude more computation time, will be needed occasionally. There are various reasons this could become desirable or necessary.

One reason could be that the "touch-up training" isn't resulting in a classification system that meets performance specifications. Another is that the overall nature of the environment may change.

As an example of the second reason, consider the malware event classifier discussed above. During the first months of 2016, exploit-kit-related activity was prevalent when compared to phishing-related activity. The latter months, however, saw a significant increase in, and prevalence of, phishing activity. This kind of shift in the malware environment would indicate a need for a system restructuring and re-optimization.

It is expected that the topological restructuring will be required much less frequently than the touch-up retraining. If the classifier is "touched up" once each day, for example, the overall Deep Swarm system re-optimization is likely to be needed only once each month or so. The relative frequencies will, of course, be application- and environment-dependent.

Computation speed is often measured in milliseconds or seconds for many PSO applications. Deep Swarm is currently, comparatively speaking, a much slower tool, with execution times measured in minutes or hours. The ten runs for which results are presented in Table III took between twenty and ninety minutes to complete in a Windows environment on a laptop computer. This is acceptable for many complex engineering applications such as the malware analysis presented here.

V. The Perl Implementation

The development team has transitioned into an Ubuntu Linux environment. Canonical swarm code was first implemented and tested in Perl.

A complete Deep Swarm implementation in Perl was completed in September 2017. The implementation was built in two scripts that cleanly represent the "inner" and "outer" swarm.

The inner swarm functionally resembles most swarm applications in use today. The distinction is, where most have user-supplied, manually-determined, internally hard-coded PSO parameters (number of particles, neighborhood size, C1/C2 and so on), our inner swarm takes command line arguments. Those can be supplied by the user manually at runtime, and can also be supplied by another calling program.

The outer swarm also resembles a typical PSO procedure, but in this case it calls the inner swarm program and treats output from there as the error function to be optimized.

It is useful to keep these two layers in separate scripts for ongoing development, loosely coupled by only sharing parameters when the outer calls the inner. Inner swarm applications can be swapped out or cycled through several variations by the outer swarm, where various inner swarms might point to different data sets or have other specialized constraints that don't concern the outer swarm.

Conversely, outer swarm experiments from various manual parameter tunings can be conducted while all are aimed at the same, consistent inner swarm. For all the auto-tuning that is now nicely being applied to the inner swarm, the outer swarm still needs those manual parameter choices made.

We have briefly considered how *another* outer swarm would behave, wrapped around and optimizing the first outer swarm, but believe it would most likely bring diminishing returns. Early runs with the outer swarm have shown fast convergence at 10 to 20 generations, further *meta* optimization there seems unnecessary, although that could be borne out by experiment.

Eleven parameters of the inner swarm are simultaneously evolved by the outer swarm:

- number of particles
- number of neighbors
- V max
- neural network weight max
- initial neural network weight max
- C1
- C2
- inertia weight start
- inertia weight end
- neural network sigmoid transfer function slope
- number of neural network hidden PEs

This list is the result of an attempt to address most available tunable parameters. The inertia weight start and end values define a linear path for the inertia weight to change during inner swarm operation.

VI. Moving Forward

The system will next be ported to Python, and then to ANSI C, each for use in a Linux environment. This environment facilitates the development and testing of a production system.

Much larger datasets are currently being analyzed. The sizes of these datasets range from 20,000 to hundreds of thousands of patterns each. Note that the production system will have to process approximately 10,000 patterns per hour.

Additional capabilities will be added to the existing Deep Swarm system. The next major development will be nesting a fuzzy rule base inside the outer shell swarm, enabling optimization of fuzzy rules, and fuzzy parameters such as fuzzy membership types, locations and slopes simultaneously with parameters of PSO and neural network structures. This will open up Deep Swarm to linguistic variables.

The coding effort is occurring in parallel with development of a Computational Swarm Open Library being established on Github at https://github.com/computationalswarm. Several hundred megabytes of source and executable code, pattern databases, and publications are available.

The public is encouraged to access this library, and to provide material for inclusion.

VII. Conclusions

A new generation of PSO, Deep Swarm, has been developed that automatically evolves optimal or near-optimal values for parameters of the PSO algorithm such as population size and neighborhood size, and, if used, parameters of associated neural network(s), fuzzy rule bases, etc.

Deep Swarm significantly reduces development time and effort for complex systems that must frequently re-optimize in highly dynamic environments.

ACKNOWLEDGMENT

We thank Matt Jonkman and Fran Trudeau at Proofpoint for their support, and for their cooperation and expertise in classifying and reclassifying malware patterns. R. C. E. thanks Jim Kennedy for his inspiration and inputs.

REFERENCES

[1] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," Proc. IEEE Intl. Conf. on Neural Networks, (Perth, Australia), IEEE Service Center, Piscataway, NJ, 1995, IV:1942-1948.

[2] R. C. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," Proc. Sixth Intl. Symposium on Micro Machine and Human Science, (Nagoya, Japan) IEEE Service Center, Piscataway, NJ, 1995, 39-43.

[3] Y. Shi and R. C. Eberhart, "Parameter selection in particle swarm optimization," Proc. Evolutionary Programming VII: Proc. EP98, Springer-Verlag, New York, 1998, 591-600.

[4] Y. Shi and R. C. Eberhart, "A modified particle swarm optimizer," Proc. IEEE Intl. Conf. Evolutionary Computation, IEEE Press, Piscataway, NJ, 1998, 69-73.

[5] R. C. Eberhart and Y. Shi, Computational Intelligence: Concepts to Implementations. Boston, MA: Elsevier (Morgan Kaufmann imprint), 2007, 87-92.

[6] K. R. Harrison, A. P. Englebrecht, and B. M. Ombuki-Berman, "The sad state of self-adaptive particle swarm optimizers," Proc. IEEE Congress on Evolutionary Computation, IEEE Press, Piscataway, NJ, 2016, 431-439.

[7] E. T. van Syl and A. P. Englebrecht, "Comparison of self-adaptive particle swarm optmizers," in Proc. 2014 IEEE Symposium on Swarm Intelligence, IEEE Press, Piscataway, NJ, 2014, 1-9.

[8] Matthew Jonkman, Proofpoint, Inc., Personal Communication (2016).

[9] D. Smilkov, "Open sourcing the embedding projector: a tool for visualizing high-dimensional data," Google Research Blog, https://research.googleblog.com/2016/12/open-sourcing-embedding-projector-tool.html, Dec. 7, 2016.

[10] W. R. Miller, D. J. Groves, A. Knopf, J. L. Otte, and R. D. Silverman, "Word Adjacency Graph Modeling: Separating Signal From Noise in Big Data," Western Journal of Nursing Research, 39:1, 2017, 166-185.