Lab 4:
Evolutionary Computation and Swarm Intelligence

2021-03-12

Introduction

Read through these instructions carefully in order to understand what is expected of you. Read the relevant sections in the course book. During the lab you are advised to create MATLAB scripts in order to easily repeat the steps for other data sets or parameters. If anything is unclear or if you have questions, please do not hesitate to contact us either:
- during the lab
- by sending us a message on Studium
- by sending us an e-mail:

- gustaf.borgstrom@it.uu.se
- chencheng.liang@it.uu.se
- nadezhda.koriakina@it.uu.se

This is the fourth (and last) of four labs in this course. You are expected to work as a team with the lab, meaning that each step is performed as a group. It is therefore *not* allowed to split up the work. Start by *skimming through the instructions* to get a feel of what will happen and then work through it from beginning to end. Make observations and take notes and it is usually a good idea to try and formulate the answers to questions right away. However, sometimes it is also useful to go back and revisit questions if you clear out a previous misconception.

About covid-19

Due to pandemic regulations, the course as a whole is given remotely. As such the lab hours are conducted over Zoom. See Studium for details about date and time, as well as the Zoom link. We use the Announcements functionality from

Studium to inform everyone about anything related to the course, so be sure to check your Studium settings so that Announcements are enabled. For general course info, see the course dashboard on Studium.

Goal of this lab

The goal of this assignment is to learn about two population methods: Genetic Algorithms (GA) and Particle Swarm Optimization (PSO). These are both methods, that work by using differently sized populations of solutions to explore large parts of the search space in parallel. Although they can be used to solve similar or identical problems, the approach of each is quite different. You will use both GA and PSO to solve several optimization problems that are challenging for gradient descent methods.

What to hand in

Hand in a report according to the instructions found on Studium. It should contain answers to all questions, and all requested plots.

Files

You will need the files `gasolver.zip` and `psopt.zip` found on Studium. Place them in your MATLAB working directory.

Task 1: Genetic Algorithms

In this section, we experimentally explore Genetic Algorithms. You will need the file `gasolver.zip`, which you can find on Studium. Unpack it into your MATLAB working directory and **add the `gasolver` directory to the matlab path (by right-clicking the directory and selecting "Add to Path")**.

Finding the largest area

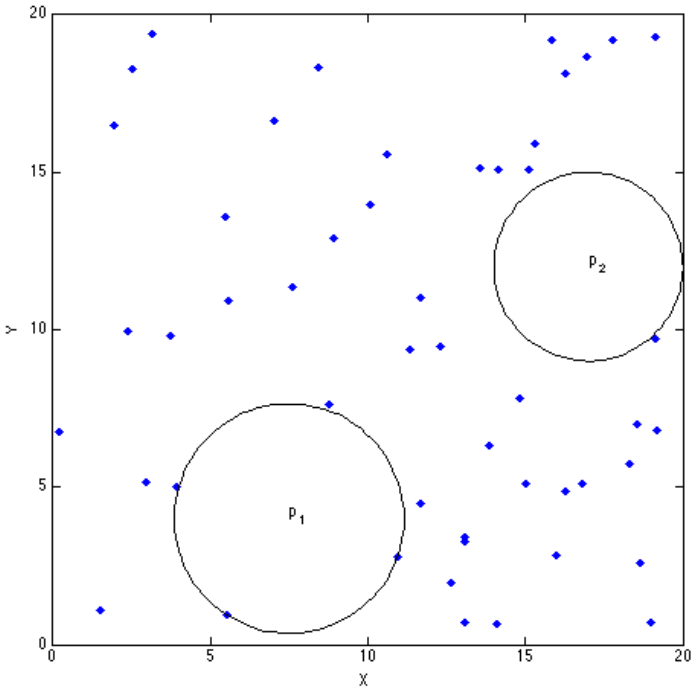

Figure 1: Two points, p_1 and p_2, with circles representing the distance to the nearest "star".

For a random scattering of points, which we will call "stars", in a bounded area, we wish to find the largest circle that can be drawn inside those same bounds that does not enclose any stars (see Figure 1). Solving this problem with a genetic algorithm requires deciding how to encode as a genome information sufficient to represent any solution. In this case, the only information we need is the center point of the circle, so our genome of point p_i will look like (x_i, y_i), representing the Cartesian coordinates.

Question 1: *Think about what each of the genetic operators means for this*

simplistic genome. Geometrically speaking, what would a 1-point crossover look like in this case? What about mutation?

For the fitness of a genome, we will use the radius of the largest circle that can be drawn at the coordinate it represents. Since the circle cannot contain any of the stars, that radius is the distance to the nearest star:

$$circle_{obj}(p_i) = \min_{s \in stars} \left\{ \sqrt{(x_i - x_s)^2 + (y_i - y_s)^2} \right\} \tag{1}$$

Question 2: *Would you expect more improvement in this problem to result from crossovers or mutations? Why? Is that what you would normally expect?*

Locate the file `lab4.mat` in the MATLAB file browser, and double click it to add the contents of several variables to you workspace. Now, you can set up the circle problem and run the GA solver like this:

```
star = GAparams
star.objParams.star = star1;
[best, fit, stat] = GAsolver(2, [0 20 ; 0 20], ...
    'circle', 50, 100, star);
```

GAsolver has three required parameters, and several optional parameters:

- The length of the chromosome (in this case, 2, for our two coordinates).
- An $n \times 2$ matrix of lower and upper bounds for each dimension. If there is 1 column, every position in the chromosome gets the same bounds; alternately, you may specify bounds for each position independently.
- A MATLAB function to calculate the fitness of the population. The function name is formed by prepending `ga_fit_` to the string you pass here; so, in this case, we are using the function `ga_fit_circle`, which you can find in your workspace. This function will be passed matrices holding the population (one genome per row) and the bounds of the variables, and an object of other parameters.
- The size of the population (default is 500).
- The number of generations (default is 1000).
- A structure of parameters for the solver. The `GAparams` struct has default values you can start with, and Table 1 gives brief explanations.

The function returns three variables:

- The chromosomes of the best individuals found. Each time a new global best is located, it is added to the top of this matrix, so `best(1,:)` is always the best individual found.
- The fitness values associated with the best individuals (`fit(1)` is the best fitness encountered).
- A structure of various statistics of the run. This is used to plot performance after the fact.

Run GAsolver now. You should get a list of the minimum, average and maximum fitness in each generation; once all generations are complete, you will also see the genome of the individual with the highest fitness.

To get a better idea of what the solver is doing, we will supply it with a visualization function that will plot some information from intermediate generations. We will do this by adding the name of the visualizer function to the parameter structure:

```
star.visual.active = 1;
star.visual.func = 'circle';
[best, fit, stat] = GAsolver(2, [0 20 ; 0 20], ...
                             'circle', 50, 100, star);
```

Run GAsolver again. As the solver runs, the red circles represent the coordinates of the population of the current generation, whereas the bold red circle shows the objective value of the best individual. Run it several more times and observe how the spread of the population over the solution space changes as the population evolves. Odds are that it won't perform terribly well (a few of the default parameters are not particularly good for this problem). To this point, you've only been running for 100 generations, so one possibility is to give the solver more time to improve the solution; however, extra generations are unlikely to help in this case. To see why, we will take a look at a plot of a measure of the diversity of the population over time.

```
ga_plot_diversity(stat);
```

Plot 1: *Include a plot showing the change in diversity of the population by generation.*

Question 3: *What are the possible sources of population diversity when using genetic algorithms? How are those sources of diversity reflected in the shape of the diversity curve in your plot?*

Table 1: Parameters for GAsolver. Default values shown in bold.

param.	values	
geneType	**'float'**	
	'binary'	$0, 1$
crossover.func	**'1point'**	
	'npoint'	set crossover.n for number of points
	'uniform'	
	'arithmetic'	(float) weighted average of n parents
	'blend'	(float) stochastic weighted average of 2 parents
	'geometric'	(float) negative values result in imaginary offspring!
	'linear'	(float) 3 offspring on a line through parents, keep best 2 (requires extra fitness evals, but more exploration)
	'none'	no crossover operator
crossover.weight	$[0-1, \dots]$	vector of parent weights (size sets # of parents)
crossover.prob	$0-1$	chance that given parents produce an offspring (default is .9)
mutate.prob	$0-1$	default is 0.005
mutate.step	$0-$	standard deviation of distribution for step size (used in some floating point mutation ops)
select.func	**'proportional'**	selection based on actual fitness values
	'rank'	selection based on ranking of fitness values
	'tournament'	pick best from pool of n parents
	'random'	ignore fitness, pick at random
select.sampling	**'roulette'**	chance of selection proportional to fitness
	'stochastic'	number of offspring proportional to fitness [**?**, section 8.5.3]
select.size	$\mathbf{2}-$	(tournament) size of pool (default = 2)
select.pressure	$1-\mathbf{2}$	(rank) selection pressure (default = 2)
scalingFunc	**'none'**	use raw fitness values
	'sigma'	scale fitness based on variance (per generation)
scalingCoeff	$0-\mathbf{1}$	less selection pressure at high values
replace.func	**'all'**	generational GA
	'worst'	replace worst individuals of current gen
	'random'	offspring replace random individuals
replace.elitist	**true**, false	fittest chromosome always survives
visual.active	Boolean	visualize the objective function
visual.func	string	name of the visualization function
visual.step	> 0	number of generations between updating the visualization
verbose	**true**, false	provide feedback in every generation
stop.func	**'generation'**	stop at maximum generation count
	'improvement'	stop when best fitness stops improving
	'absolute'	stop when a (known) optimum is reached
stop.window	default: 100	generations to wait for fitness improvement
stop.direction	'max','min'	whether to maximize or minimize the objective
stop.value	default: 1	known best fitness value
stop.error	default: .005	how close to get to optimum before stopping

Question 4: *Why is population diversity important?*

To try and maintain diversity a little longer, we will modify some of the parameters to the algorithm. You can get an overview of the current settings by typing:

```
ga_show_parameters(star);
```

One thing that can affect diversity is the choice of a selection heuristic. The default here, which you have been using up to now, is *fitness* selection (also called *proportional*, or *roulette wheel* selection), but, as you no doubt recall from lecture, *rank* based selection is often a better choice for avoiding premature convergence.

Question 5: *What's the difference between rank and fitness selection? Why does that make rank based selection better at avoiding premature convergence?*

In GAsolver, choices like selection heuristics, crossover and mutation operations, and replacement strategies are all implemented as MATLAB functions. To select a different function, you supply its name as a parameter, like this:

```
star.select.func = 'rank';
```

Rank selection also makes use of the selection pressure parameter, a number between 1 and 2 which affects the likelihood of more fit individuals being selected: at a value of 2, high fitness individuals are much more likely to be selected than low fitness individuals, while at 1 there is a much smaller difference in the probabilities.

Plot 2: *For each pressure value of 2, 1.75, 1.5, 1.25 and 1 provide one plot of the population diversity over 100 generations using rank based selection.*

Question 6: *What selection pressure results in the most promising looking diversity curve? Run the algorithm several times using that pressure setting. Report the fitness value and position you were able to locate?*

The replacement strategy determines which individuals from the old population and the new pool of offspring survive into the next generation. The default here is a *generational*, meaning that the offspring completely replace the old population, and *elitist*, meaning that the best individual encountered so far is always preserved unchanged.

Question 7: *What are the possible downsides of using elitism?*

Now try the solver on two other star maps. `star2` has a global optimum in an area of low density which also has a couple of strong local optima; in `star3` the global optimum is found in the middle of an otherwise high-density section. Pass each of these maps to the solver by setting `star.objParams.star` equal to the new map, and then running the solver again, using the settings that have resulted in the best performance so far.

Question 8: *What settings did you use? How well did the solver perform on the more difficult maps? Explain any difference in performance you observe.*

Minimizing a Function

Now you will use genetic algorithms to minimize Ackley's function, a widely used test function for global minimization. Generalized to n dimensions, the function is:

$$\begin{gathered} f(x) = -a \cdot e^{-b \cdot \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n} \cdot \sum_{i=1}^{n} \cos(cx_i)} + a + e^1 \\ \text{for} \quad -32.768 \leq x_i \leq 32.768, \quad i = 1, \ldots, n \end{gathered} \tag{2}$$

We will use the standard parameters $a = 20, b = 0.2, c = 2\pi$. Ackley's function has several local minima, and a global minimum of $f(x) = 0$, obtainable at $x_i = 0$ for $i = 1, \ldots, n$. Take a moment to familiarize yourself with the shape of the two-dimensional Ackley's function, using the supplied visualizer function:

```
ack = GAparams;
ack.visual.type = 'mesh';
ga_visual_ackley([],[],[],[],[],[],ack.visual,[],[]);
```

This function is challenging partly because of the strong local minima that surround the global minimum. To get a better look, you can change the bounds of the surface plot as follows:

```
ack.visual.bounds = [-2, 2];
ack.visual.interval = 0.05;
ga_visual_ackley([],[],[],[],[],[],ack.visual,[],[]);
```

You can also change the plotting function to use many of the 3-D plots supplied by MATLAB , by substituting the name of the function ('`mesh`') speci-

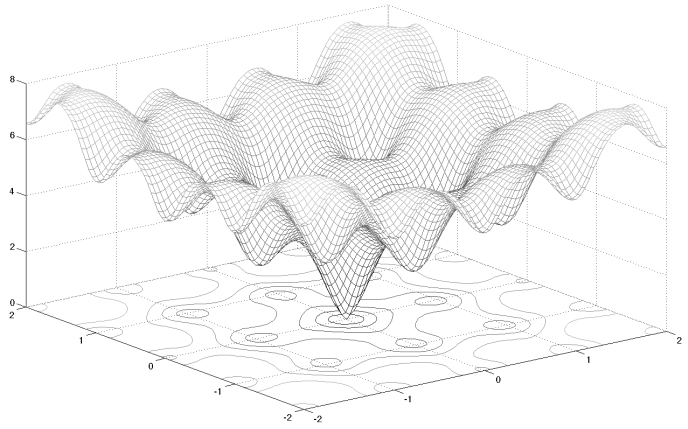

Figure 2: The area around the global minimum for Ackley's function in 2-D.

fied in `ack.visual.type`. Examples include '`contour`', '`surf`', '`surfc`', etc. Just remember that any adjustments you make to `ack.visual` will also affect the visualization you get when you run the solver.

Question 9: *Why would the global minimum be difficult to find using a gradient descent method?*

Set up the problem for solving the 20 dimensional Ackley's function:

```
ack.stop.direction = 'min';
ack.visual.func = 'ackley';
ack.visual.active = true;
[best, fit, stat] = GAsolver(20, [-32, 32], ...
   'ackley', 200, 250, ack);
```

This can be a difficult function to minimize, and the initial settings may not work well. You can try to modify the selection or replacement strategies, as we did in the last section, but this time we will also try a few other approaches.

We will experiment mainly with three different settings: *crossover operation*, *mutation decay* and *replacement strategy* (all options are listed in Table 1).

The 1-point, n-point, and uniform *crossover operations* operate by swapping the values of some genes between the parents. This is how binary encoded crossover usually works, but it's also valid for floating point encodings like we're using here. The arithmetic, blend, and linear operations are specifically intended for floating point encodings, and all generate offspring that are some weighted

linear combination of the parents.[1] The effectiveness of the float operators depends on the `weight` parameter, a vector of length n that gives weights to each parent (and, incidentally, determines the number of parents). The blend operation (sometimes called BLX-α) is slightly different: it always takes 2 parents, it takes a single weight parameter (0.5 is the default), and it generates a pair of weights stochastically for each position.

The mutation operator can also have a profound effect on performance. There are only two operators provided here, uniform and inorder (the latter restricts possible mutations to a randomly selected region of the gene). However, there are a couple of different strategies for dynamically adjusting the mutation rate. The first strategy is to start with a large mutation rate, and decrease it over time. This strategy is controlled by setting the `mutate.decay` parameter to `'none'`, `'linear'`, or `'exponential'`. The second strategy is to base the probability of mutation on the fitness of the individual, where less fit individuals are more likely to mutate. This is controlled by setting `mutate.proportional` to true or false.

In generational GA, the offspring replace the parents each generation, how to do this is governed by the *replacement strategy*. So far, we've been replacing all the parents except for the best individual, but this might not be the best strategy. Another option is to only take offspring that have a better fitness. We will call this *comparative* replacement (there doesn't seem to be a commonly accepted name for it). You can activate it by setting `replace.comparative` to true. Try comparative replacement now, using a few of the more promising combinations of parameters you've encountered up to this point.

Question 10: *Explore at least ten different combinations of settings in total, make sure each setting is run several times. Write down (e.g., in a table), for each set-up what parameters you used, the average fitness you achieved. Motivate your choice of explored parameter combinations.*

Question 11: *Try using GAsolver to find the minimum for the 100 dimensional Ackley's function, using the best parameters you found in the previous question. How well does it perform now and how can you improve the performance?*

Training a Neural Network using GA

One widely used application domain for both GA and PSO is the tuning of parameterized systems. You've been dealing with a parameterized system in this course: neural networks. Training a neural network means adjusting the weights

[1] The geometric crossover is also a weighted average of multiple parents, but it generates imaginary values when the search space includes negative numbers.

of the inputs; every weight is a parameter of the system. The objective function for a neural net is the measure of the network's error (mse, sse, etc.).

One popular technology for training neural networks using GA is NeuroEvolution of Augmenting Topologies (NEAT). We will not experiment with it practically here, but instead watch a short video clip and browse an article. We begin by watching a video demonstrating the result of trying to play Super Mario World using NEAT. Watch the video at `https://www.youtube.com/watch?v=qv6UVOQ0F44`.

Question 12: *What are the inputs and outputs of the networks evolved in the video.*

Skim the following article: `https://www.soa.org/globalassets/assets/Library/Newsletters/Forecasting-Futurism/2013/december/ffn-2013-iss8.pdf`

You might also find the original paper introducing NEAT of interest: `http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf`

Question 13: *What is the major challenge when doing crossover between neural networks, which NEAT addresses, apart from the fact that weights in one network are unlikely to mean the same thing in another?*

Task 2: Particle Swarm Optimization

In this section, we experimentally explore Particle Swarm Optimization.There are a number of Particle Swarm toolboxes available for MATLAB , although at this time there are none that are very polished. For this task, you'll be using an open source PSO toolbox modeled after the MATLAB Optimization Toolbox. You will need the file `psopt.zip`, which you can find on Studium. Unpack it into your MATLAB working directory, and **add both the `psopt` directory and its sub-directories to the `matlab` path (by right-clicking the directory and selecting "Add to Path")**.

Minimizing a Function

We will start by exploring Ackley's function once more.

```
options = psooptimset('DemoMode','fast',...
        'PlotFcns',{@psoplotswarmsurf,@psoplotbestf},...
        'PopInitRange',[-32;32],'InertiaWeight',1,...
```

```
        'SocialAttraction',2,'CognitiveAttraction',2);
[x, fval] = pso(@ackleysfcn,20,[],[],[],[],[],[],[],options);
```

The first line creates a structure of parameters. You can view the available parameters by typing `psooptimset` with no arguments or return values. When you create an options structure like `options` here, you only need to specify fields and values that you want to differ from the defaults. You can pass another options structure to `psooptimset` as well, which will copy the old structure and then apply any changes. For example, to change only the number of generations specified by `options`:

```
options = psooptimset(options, 'Generations', 500);
```

The `pso` function takes a handle to the objective function, the number of dimensions in the problem, and the options structure. The empty arguments in the middle could be used to specify linear and nonlinear constraints, but we will ignore them.

When you run the swarm, you'll get a window with the two plots specified above. One shows the particle positions and fitness surface of the function (in the first two dimensions of the problem only). The other shows the best and average fitness scores in each iteration.

In this first trial, we've used settings that replicate what you could call "classical" PSO. The resulting swarm behavior demonstrates one of the biggest problems with PSO algorithms, a phenomenon known as *swarm explosion*: the velocities of the particles steadily increase until the swarm flies apart. There have been lots of methods proposed to deal with swarm explosion over the years, and we will investigate a few right now.

The simplest is *velocity clamping*: we impose a maximum velocity in each dimension, and any time a particle's velocity exceeds that maximum, we reset it to the maximum and continue. To try velocity clamping, you need to set the parameter `VelocityLimit` to $\delta(\max_i - \min_i)$ in each dimension, where $\delta \in (0, 1]$. For this problem, there are no real differences between the dimensions, so you can pass a single velocity limit and it will be applied to all dimensions. Try clamping with velocity limits of 30, 15, 5, and 1. In order to better see the effect of velocity clamping, add `@psoplotvelocity` to the list of plotting functions.

Question 14: *What happens to the maximum velocity of the swarm over time when using velocity clamping? How does the maximum velocity compare with the theoretical velocity limit (the red line at the top of the velocity plot)?*

A more interesting approach to controlling swarm explosion is by introducing *inertia weights*. Inertia in a PSO refers to the amount of influence a particle's

current velocity has on the particle's next velocity. To understand the inertia weight, we need to consider the velocity equation for global best, or *gbest*, PSO (the effect on *lbest* and *pbest* is similar).

$$v_{id}(t) = wv_{id}(t-1) + c_1 r_1[p_{id} - x_{id}(t-1)] + c_2 r_2[p_{gd} - x_{id}(t-1)] \quad (3)$$

Here $v_{id}(t)$ is the velocity of particle i in dimension d at time step t, $x_{id}(t)$ is the position of that same particle, p_{id} is the particle's personal best position, and p_{gd} is the global best position. The constants c_1 and c_2 are positive acceleration factors controlling the particles' cognitive and social components: high c_1 values cause the particle to have increased confidence in its own previous best, while high c_2 values cause the particles to have increased respect for the group's best position. You can change the values of c_1 and c_2 with the `CognitiveAttraction` and `SocialAttraction` options, respectively. Frequently the values assigned to these constants are somewhere around 2.

The inertia weight is another constant, w, controlled by the option `InertiaWeight`. Up to now we've used a value of $w = 1$, which replicates the *gbest* velocity equation without inertial weight. Setting $w \geq 1$ causes the particles to accelerate constantly, while $w < 1$ should cause the particles to gradually slow down. By carefully balancing the three parameters w, c_1 and c_2, it is possible to fine tune the algorithm, balancing both the social and cognitive components, and the tension between exploration and exploitation. There are a lot of suggested combinations for these values in the literature; for example $c_1 = c_2 = 2$ and $w = 1$, or $c_1 = c_2 = 1.49$ and $w = 0.7968$. Try those combinations now, as well as a few others, and see how well behaved the resulting swarms are. Be sure to try both with and without velocity clamping turned on.

Question 15: *What values for w, c_1 and c_2 worked best on this problem? How close did this swarm come to locating the global optimum?*

Question 16: *Was velocity clamping still necessary to prevent swarm explosion, or were you able to find a combination of values that kept the swarm together?*

Another very common modification is to let the inertia weight decay over time. To cause the weight to linearly decay, set the inertia weight to a vector $[w_{\max}; w_{\min}]$. A frequently used range is $[0.9; 0.4]$. Try switching off velocity clamping, and see if you can get the inertia weight set to some combination of values that reliably comes close to the optimum, and avoids the swarm explosion.

Question 17: *What was the best combination of decaying inertia weights, and social and cognitive coefficients? What was the best fitness value you found using that combination?*

At this point, it should be clear that the right combination of inertia weight, social and cognitive constants, and velocity clamping can yield a pretty reliable particle swarm. There is one more quite interesting method for avoiding swarm explosion that we should consider, however, which is called *constriction*. The constriction velocity equation for *gbest* is seen in Equation 4.

$$v_{id}(t) = \chi[v_{id}(t-1) + \phi_1 r_1(p_{id} - x_{id}(t-1)) + \phi_2 r_2(p_{gd} - x_{id}(t-1))] \quad (4)$$

χ is called the *constriction coefficient*. Letting $\phi = \phi_1 + \phi_2$, we further require that $\phi > 4$, and then define χ in terms of ϕ.

Question 18: *What is the equation for the value of the constriction coefficient in terms of ϕ?*

The advantage of constriction is that it does not require velocity clamping. As long as the parameters are selected as described, the swarm is guaranteed to converge. Unfortunately, the toolkit we are using does not include constriction. We can, however, implement constriction using the inertia weight approach. By choosing the right values of w, c_1 and c_2, we can make Equation 3 be equivalent to Equation 4 for a specific ϕ_1 and ϕ_2.

Question 19: *Consider the constriction equation with $\phi_1 = 4$, $\phi_2 = 2$. What is the constriction coefficient for these values? What values of w, c_1 and c_2 would we have to use in Equation 3 in order to implement constriction with these values?*

Try constriction now for $\phi_1 = 4$, $\phi_2 = 2$, using the appropriate parameters in the inertia weight model.

Question 20: *Describe the behavior of the swarm when using constriction. Does it locate the global optimum? How quickly does it converge?*

Training a Neural Network using PSO

Previously you saw how to train neural networks using genetic algorithms. Here we will make a similar study on how to use PSO instead. In constrast to using NEAT, we will keep the structure here static, and we will start by training a network to solve the XOR problem from the first lab. For a network with a given topology, the error depends only on the weights—in other words, the set of weights is a sufficient encoding of the solution space.

Gradient descent and related learning rules for neural nets are *local* optimization algorithms: from a single starting state (i.e. random weights assigned in

initialization), the algorithm searches for an optimum, employing various strategies to avoid getting stuck in local optima. Recall Lab 1, when you trained an XOR network: from some start positions, gradient descent led to a network that encoded one of the two possible solutions, while from other start positions it found a non-solution that represented a local minimum or plateau. PSO is *global* optimization: it searches larger parts of the solution space. As a result, it should be less likely to get stuck.

Just like in lab 1, we start by creating the network:

```
global psonet P T
P = [[0; 0] [0; 1] [1; 0] [1; 1]]
T = [0 1 1 0]
psonet = newff(P, T, [2], {'tansig' 'logsig'}, 'traingd', ...
               '', 'mse', {}, {}, '')
```

Make sure you name the variables exactly as given above. We're defining these variables as global so that the fitness function will be able to see the net you've created, but it only works if the names are exactly the same (sometimes programming in MATLAB is just like that). Also, we've specified a training algorithm for the network here because MATLAB expects one, but we won't actually use it to train the network.

The fitness function we will be using is `nntrainfnc`, and you can find it in the `testfcns` directory. Take a look at the function, it's quite straightforward. For each particle, it sets the weights of the network to the values of the particle's current position. Then it calculates the error using those weights for all the test patterns. The error is the fitness value that we wish to minimize. Unfortunately, this is all implemented using fairly high-level access to the neural network's structure, so the PSO is going to run quite a bit slower. You probably will want to compensate by turning the number of particles down (something in the range of 5-10 should work well).

Here's the basic setup for training the network:

```
options = psooptimset('PopInitRange',[-1;1],...
 'PopulationSize',5,'DemoMode','pretty',...
 'PlotFcns',{@psoplotswarm,@psoplotbestf,@psoplotvelocity});
[x, fval] = pso(@nntrainfcn,9,[],[],[],[],[],[],[],options);
```

Note that we are now using `psoplotswarm` instead of `psoplotswarmsurf`.

Question 21: *Given the network topology we used in Lab 1 and the problem*

definition above, we've defined the problem as having 9 dimensions. What possible explanation is there for that?

After training, you can evaluate the best solution the PSO found like this:

```
psonet = setx(psonet, x);
plot_xor(psonet);
```

Naturally, you may need to make a few parameter adjustments before you get a good solution.

Plot 3: *Include the output from* `plot_xor` *for the best solution you were able to locate.*

Question 22: *How does the plot of the PSO solution compare with the plots you got in lab 1? Why might these plots be different?*

Task 3: Wrapping up

We would encourage you to help us out by giving us some feedback on this lab. We appreciate any input *a lot*. The first question is regarding the *lab content*, while the other is about the *lab assistance format*. Please take a moment to briefly answer them. Any feedback is highly appreciated!

Question 23: *How well did this lab help to understand concepts from the lectures? Is there a specific concept that the lab has helped to clarify for you? Is there a concept from the relevant lectures that should have been covered more?*

Question 24: *If you attended the online lab: please tell us if there was anything that you liked/disliked in general (with focus on concrete things we could consider and improve).*