Chapter 13 The Benchmarker

Abstract In this chapter we add the benchmarker process which can sequentially spawn population_monitors and apply them to some specified problem/simulation. We also extend the database to include the *experiment* record, which the benchmarker uses to deposit the traces of the population's evolutionary statistics, and to recover from crashes to continue with the specified experiment. The benchmarker can compose experiments by performing multiple evolutionary runs, and then produce statistical data and GNUplot ready files of the various evolutionary dynamics and averages calculated within the experiment.

Though in the previous chapter we have completed the development of the most important part of keeping track of the population's statistics and progress, we can still go a step further and add one more program, the *benchmarker*. When running a simulation or experiment, the progress of the population, the trace, represents a single evolutionary path of the population. When analyzing the functionality of our system, when we want to benchmark a new added element, we might wish to run the simulation multiple times, we might want to create multiple traces for the same problem, and then average them before starting to analyze the functionality of our TWEANN, or the results of applying it to some simulation or problem.

The *benchmarker* process we want to create here is in some sense similar to the one we implemented in Section-7.7. This program will offer us a concise and robust way in which to apply the population_monitor to some problem multiple times, and thus build a dataset by averaging the performance of our neuroevolu–tionary system from multiple applications to the problem, from multiple evolutionary runs. The benchmarker will be called with the following parameters:

1. The *INIT_CONSTRAINTS* parameter, which will specify the type of problem the benchmarker will create the populations for.
2. The parameter N, which will specify the number of times the benchmarker should apply the neuroevolutionary system to the problem.
3. The termination condition parameters (evaluations limit, generation limit, and fitness goal).

The benchmarker's operational scenario would be as follows: The benchmarker process would first spawn the population_monitor. Then wait for the population_monitor to reach its termination condition, send benchmarker the accumulated trace record, and then terminate. Afterwards, the benchmarker would store the trace into its trace accumulator, and spawn a new population_monitor which would try to solve the problem again. This would continue for N number of times, at which point the benchmarker would have accumulated N traces. It could then average the trace results and form a single trace average (the various averages

G.I. Sher, *Handbook of Neuroevolution Through Erlang*,
DOI 10.1007/978-1-4614-4463-3_13,

between all the traces composing the experiment). This trace average can then be written to file in the format which can be graphed and visualized, by perhaps a program like gnuplot [1].

In the following sections we will implement this *benchmarker* process. The ability to determine and graph the performance statistics of a neuroevolutionary system allows one to advance it, to see where it might have flaws and what new features should be added, and the affect of those new features on its performance. The benchmarker program also assists in conducting research, for the results and applications of the neuroevolutionary system must be presented at one point or another, and thus a benchmark of the neuroevolutionary system's general and average performance on some task must be composed. The experiment must be run multiple times, such that the accuracy and the standard deviation of the results can be calculated. And that is exactly what the *benchmarker* program will assist in doing.

13.1 The benchmarker Architecture

The purpose of the benchmarker process is simple, to spawn a population_monitor, wait for it to finish solving the problem or reach a termination condition and send its composed *trace* to the benchmarker process (if the benchmarker was the one that spawned the population_monitor), and then respawn another population_monitor, repeating the procedure N times. Once the benchmarker has done this N number of times, and thus has accumulated N traces, the benchmarker is to analyze the traces, build the averages of those traces, and write this data to a file, and optionally print it to console.

Because gnuplot is so prevalent in plotting data in the scientific community, we want the benchmarker to write to file the resulting benchmark data in a format that can be directly used by gnuplot. Some of the information that can be plotted is: *Fitness Vs. Evaluations, NN Size Vs. Evaluations*, and *Specie Diversity Vs. Evaluations*.

Furthermore, assume that we are running our benchmark on a single machine. We planned on applying our neuroevolutionary system to some problem 100 times, each for 100000 evaluations. And on the 90^{th} evolutionary run there is a power outage, and we lose all 90 evolutionary run traces when we only had 10 more to go before completing the full experiment composed of 100 evolutionary runs. To prevent such situations, we must of course save the trace results which belong to the same experiment, after every evolutionary run. Thus if there is a power outage, or we wish to stop the experiment at some point, we need to ensure that whichever evolutionary runs have already been done, will have their traces backed up, and thus give us a chance to continue with the experiment when we are ready again.

To add such functionality, we will create a new mnesia table called *experiment*, which will allow for every experiment to have its own id or name, and a trace_acc list where it will accumulate the traces which belong to that particular experiment. It will be the benchmarker process that will backup the traces to their appropriate experiment entry, after every completed simulation or problem run.

To accomplish all of this, the benchmarker process needs to be able to do the following tasks:

4. Know how many evolutionary runs to perform for the experiment.
5. Know the name of the experiment, so that it can store the traces to their appropriate locations in the mnesia table.
6. Be able to specify the initial state parameters with which to start the population_monitor process, and restart it after a crash.

This means that other than adding the *experiment* record to the records.hrl file and creating a mnesia table of the same name, we must also modify how the population_monitor is started. Currently, it uses the macros defined within the module. These macros define how large the initial population size should be, the termination conditions... This makes it difficult to start the population_monitor from another module, and control the population_monitor's parameters from the same. Thus we will need to expand its *state* record to include the previously macro defined parameters, and add a new function with which to start the population_monitor, a function which can be executed with a list of parameters, the parameters that are then entered into the state tuple with which the population_monitor is started.

In the following sections we create the new records and add the new table to the mnesia database. We then make a small modification to the population_monitor module, move the previously macro defined parameters into the state record, and add a new function with which the population_monitor can be started and have its *state* record initialized. Finally, we then create the actual benchmarker module.

13.2 Adding New Records

We need to modify the population_monitor's state record, and then add two new records to the records.hrl file. The population_monitor's new state record will include all the elements that were previously defined through the macros of that module. With regards to the two new records to be added to the records.hrl, one of them will be the new mnesia table, *experiment,* and the other record, *pmp* (population monitor parameters) will be used specifically by the benchmarker to call and start the population_monitor process with a certain set of parameters, thus setting the population_monitor's initial state tuple to the proper values.

The population_monitor originally specified its state and other parameters for its operation using the macros and records at the top of the module, as shown in Listing-13.1.

Listing-13.1 The macros and records originally used by the population_monitor process.

```
-define(INIT_CONSTRAINTS,[#constraint{morphology=Morphology, connec-
tion_architecture=CA, population_evo_alg_f=steady_state} || Morphology<-[xor_mimic],CA<-
[feedforward]]).
-define(SURVIVAL_PERCENTAGE,0.5).
-define(SPECIE_SIZE_LIMIT,10).
-define(INIT_SPECIE_SIZE,10).
-define(INIT_POPULATION_ID,test).
-define(OP_MODE,gt).
-define(INIT_POLIS,mathema).
-define(GENERATION_LIMIT,100).
-define(EVALUATIONS_LIMIT,100000).
-define(GEN_UID,genotype:generate_UniqueId()).
-define(FITNESS_GOAL,1000).
-record(state,{ op_mode, population_id, activeAgent_IdPs=[], agent_ids=[], tot_agents,
agents_left, op_tag,agent_summaries=[], pop_gen=0, eval_acc=0, cycle_acc=0, time_acc=0,
step_size, next_step, goal_status,evolutionary_algorithm, fitness_postprocessor,
selection_algorithm, best_fitness }).
```

Because the population_monitor's macros are module specific, and we would like to be able to specify in which manner to start the population_monitor, what its fitness goal should be, evaluation and generation limits, and what polis it should use... we need to move all the macro defined elements into the population_monitor's state record. This way the benchmarker process can call the population_monitor and specify all these previously macro defined parameters. We also add one extra parameter to the state record, the *benchmarker_pid* element, which can be set to the PId of the benchmarker process, and then used by the population_monitor to send its trace to the benchmarker process that spawned it. The population_monitor's new state record is shown in Listing-13.2, where the newly added elements are shown in boldface.

Listing-13.2 The updated *state* record of the population_monitor module.

```
-record(state,{
    op_mode = gt,
    population_id = test,
    activeAgent_IdPs = [],
    agent_ids = [],
    tot_agents,
    agents_left,
```

```
    op_tag,
    agent_summaries = [],
    pop_gen = 0,
    eval_acc = 0,
    cycle_acc = 0,
    time_acc = 0,
    tot_evaluations = 0,
    step_size,
    goal_status,
    evolutionary_algorithm,
    fitness_postprocessor,
    selection_algorithm,
    best_fitness,
    survival_percentage = 0.5,
    specie_size_limit = 10,
    init_specie_size = 10,
    polis_id = mathema,
    generation_limit = 100,
    evaluations_limit = 100000,
    fitness_goal = inf ,
    benchmarker_pid
}).
```

When we start the population_monitor, we want to be able to define these elements. Their default values are shown in the state record, but every-time we run an experiment, we want to be able to set these parameters to whatever we want. Thus, we add the *pmp* (population monitor parameters) record to the records.hrl, so that it can be set by the benchmarker, and read by the population_monitor. This new record is shown in Listing-13.3, and its elements are defined as follows:

1. **op_mode**: Allows the benchmarker to define the mode in which the population_monitor operates. Thus far we only used the gt, which we have not yet used to specify any particular mode of operation, but we will in a much later chapter. In the future we can define new modes, for example the *throughput* mode during which the agents are not tuned or evaluated, but simply tested for whether they are functional, whether they can gather signals through sensors and output actions through their actuators. The *throughput* op_mode could also then be used to benchmark the speed of the cycle of the NN based agent, and thus used to test which topologies can process signals faster, and which designs and architectures and implementations of neurons, sensors, actuators, and cortexes are more efficient. Or we could specify the op_mode as *standard,* which would make the population monitor function in some standard default manner. With regards to *gt,* it stands for *genetic tuning*, but due to our not yet having specified other operational modes, or taken advantage of this parameter, it is effectively the standard mode of operation until we add a new one in Chapter-19.

2. **population_id**: Allows the benchmarker to set the population's id.
3. **survival_percentage**: Allows the benchmarker to set which percentage of the population survives during the selection phase.
4. **specie_size_limit**: Allows the benchmarker to set the size limit of every specie within the population. This is an important parameter to define when starting an experiment.
5. **init_specie_size**: Allows the benchmarker to define the initial size of the specie. For example the experiment can be started where the initial specie size is set to 1000, but the specie size limit is set to 100. In this way, there would be a great amount of diversity (given the constraint is defined in such a manner that NN based agents have access to a variety of plasticity functions, activation functions...), but after a while only 100 are allowed to exist at any one time. Or things could be done in the opposite way, the initial specie size can be small, and the limit specie size large. Allowing the specie to rapidly expand in numbers and diversity, from some small initial bottleneck in the population.
6. **polis_id**: Allows the benchmarker to define in which polis the population_monitor will create the new agent population.
7. **generation_limit**: Every experiment needs a termination condition, and the benchmarker specifies the generation limit based termination condition for the population_monitor, using this parameter.
8. **evaluations_limit**: Lets the benchmarker specify the evaluations limit based termination condition.
9. **fitness_goal**: Lets the benchmarker specify the fitness based termination condition.
10. **benchmarker_pid**: This parameter is set to undefined by default. If the population_monitor has been spawned for a particular experiment by the benchmarker, then the benchmarker sets this parameter to its own PId. Using this PId, the population_monitor can, when the neuroevolutionary run has reached its termination condition, send its trace to the benchmarker process.

Listing-13.3 The new *pmp* (population monitor parameters) record added to the records.hrl

```
-record(pmp,{
    op_mode=gt,
    population_id=test,
    survival_percentage=0.5,
    specie_size_limit=10,
    init_specie_size=10,
    polis_id = mathema,
    generation_limit = 100,
    evaluations_limit = 100000,
    fitness_goal = inf ,
    benchmarker_pid
}).
```

The *pmp* record does not necessarily need to be used only by the benchmarker. The researcher can of course, rather than specifying these parameters in the population_monitor module and then recompiling it, simply start the population_monitor using the pmp record and the new prep_PopState/2 function we will build in the next subsection, and in this way define all the necessary experiment parameters.

The new *experiment* table we will add to the mnesia database will hold all the general, experiment specific data, particularly the traces. This is the record that the benchmarker populates as it runs the problem or experiment multiple times to generate multiple traces. The *experiment* record is shown in Listing-13.4, and its elements are defined as follows:

1. **id**: Is the unique id or name of the experiment being conducted. Because we wish for this new mnesia table to hold numerous experiments, we need to be able to give each experiment its own particular id or name.
2. **backup_flag**: This element is present for the use by the benchmarker. When we start the benchmarker program with the experiment tuple whose backup_flag is set to false, it does not backup that particular experiment to mnesia. This might be useful when we wish to quickly run an experiment but not write the results to the database.
3. **pm_parameters**: This element will store the pmp record with which the population_monitor was started for this particular experiment. This will allow us to later on know what the experiment was for, and how the population_monitor was started (all the initial parameters) to produce the results and traces in the experiment entry. This way the experiment can be replicated later on.
4. **init_constraints**: Similarly to the pm_parameters which defines how the population_monitor runs, we also need to remember the parameters of the population itself, and the experiment to which the traces belong. This information is uniquely identified by the init_constraints list with which the population is created. Having the init_constraints will allow us to later on replicate the experiment if needed.
5. **progress_flag**: This element can be set to two values: *in_progress* and *completed.* The experiment is in progress until it has been run for *tot_runs* number of times, and thus the experiment has accumulated tot_runs number of traces in its *trace_acc* list. If for example during the experiment run there is a power outage, when we later go through all the experiments in the experiment table, we will be able to determine which of the experiments were interrupted, based on their progress_flag. Any experiment whose progress_flag is set to in_progress, but which is not currently running, must have been interrupted, and still needs to be completed. Once it is completed, the progress_flag is set to: *completed*.
6. **trace_acc**: This is a list where we store the trace tuples. If we apply our TWEANN to some particular problem 10 times, and thus perform 10 evolutionary runs, we keep pushing new trace tuples into this list until it contains

all 10 traces, which we can later use at our leisure to build graphs and/or deduce performance statistics.

7. **run_index**: We plan on running the experiment some *tot_runs* number of times. The run_index keeps track of what the current run index is. If the experiment is interrupted, using this and other parameters we can restart and continue with the experiment where we left off.
8. **tot_runs**: This element defines the total number of times that we wish to perform the evolutionary run, the total number of traces to build this particular experiment from.
9. **notes**: This can contain a data of any form; string, lists, tuple... This element simply adds an extra open element where some other data can be noted, data which does not belong to any other element in this record.
10. **started**: This element is the tuple: {date(), time()}, which specifies when the experiment was started.
11. **completed**: Complementary to the *started* element, this one stores the date() and time() of when the experiment was finally completed.
12. **interruptions**: This element is a list of tuples, whose form is: {date(), time()}. These tuples are generated every time the experiment has been restarted after an interruption. For example assume we are running an experiment, and on the 4th run, at which point the trace_acc already contains 3 trace tuples, the experiment was interrupted. Later on when we wish to continue with the experiment, we look through the mnesia database, in the experiment table, for an experiment whose progress_flag is set to *in_progress*. When we find this experiment, we know it has been interrupted, we take its pm_parameters and init_constraints and continue with the experiment, but also, we push to the *interruptions* list the tuple {date(),time()}, which ensures that this experiment notes that there was an interruption to the experiment, it was not a single continues run, and that though we do not know when that interruption occurred, we did continue with the experiment on the date: date(), and time: time().

Listing-13.4 The *experiment* record.

```
-record(experiment,{
    id,
    backup_flag = true,
    pm_parameters,
    init_constraints,
    progress_flag=in_progress,
    trace_acc=[],
    run_index=1,
    tot_runs=10,
    notes,
    started={date(),time()},
    completed,
    interruptions=[]
```

```
}).
```

With all the new records defined, we can now move forward and make the small modification to the population_monitor module, creating its new *prep_PopState/2* function, which will allow the benchmarker, and the researcher, to start the population_monitor process with its state parameters defined by the *pmp* record that the *prep_PopState/2* is executed with.

13.3 Updating the population_monitor Module

Instead of using the macros, we now store all the parameters in the population_monitor's state record. To start the population_monitor with a particular set of parameters, we now need to create a new function in which we define and set the *state* to the particular parameters we want the population_monitor to operate under. To set everything up for a population_monitor, we only need the parameters defined in the *pmp* and the *constraint* record. Thus we create the prep_PopState/2 function which is executed with the pmp record, and a list of constraint records, as its parameters. The new prep_PopState/2 function is shown in Listing-13.5.

Listing-13.5 The prep_PopState/2 function used to initialize the state parameters of the population_monitor.

```
prep_PopState(PMP,Specie_Constraints)->
    S=#state{
            op_mode=PMP#pmp.op_mode,
            population_id = PMP#pmp.population_id,
            survival_percentage=PMP#pmp.survival_percentage,
            specie_size_limit=PMP#pmp.specie_size_limit,
            init_specie_size=PMP#pmp.init_specie_size,
            polis_id=PMP#pmp.polis_id,
            generation_limit=PMP#pmp.generation_limit,
            evaluations_limit=PMP#pmp.evaluations_limit,
            fitness_goal=PMP#pmp.fitness_goal ,
            benchmarker_pid=PMP#pmp.benchmarker_pid
    },
    init_population(S,Specie_Constraints).
```

As can be seen, we now execute the init_population/2 function with the state tuple rather than the original *population_id* and the *opmode* parameters. This means that all the other functions which originally used the macros of this module, need to be slightly modified to now simply use the parameters which are now specified within the population_monitor's state record. The modifications are very

small and few in number, and are thus not shown. The updated population_monitor module can be found in the 13th chapter of the available supplementary material [2].

Finally, we modify the termination clause of the population_monitor, since now at the moment of termination, the population_monitor needs to check whether it was a benchmarker that had spawned it. The population_monitor accomplishes this by checking the *benchmarker_pid* parameter. If this parameter is set to *undefined*, then the population_monitor does not need to send its trace anywhere. If the benchmarker_pid is defined, then the process forwards its trace to the specified PId. The updated *terminate/2* callback is shown in Listing-13.6.

Listing-13.6 The updated terminate/2 function, capable of sending the benchmarker the population_monitor's trace record, if the benchmarker was the one which spawned it.

```
terminate(Reason, S) ->
    case S of
            [] ->
                    io:format("******** Population_Monitor shut down with Reason:~p, with
State: []~n",[Reason]);
            _ ->
                    OpMode = S#state.op_mode,
                    OpTag = S#state.op_tag,
                    TotEvaluations=S#state.tot_evaluations,
                    Population_Id = S#state.population_id,
                    case TotEvaluations < 500 of
                            true ->%So that there is at least one stat in the stats list.
                                    gather_STATS(Population_Id,0);
                            false ->
                                    ok
                    end,
                    P = genotype:dirty_read({population,Population_Id}),
                    T = P#population.trace,
                    U_T = T#trace{tot_evaluations=TotEvaluations},
                    U_P = P#population{trace=U_T},
                    genotype:write(U_P),
                    io:format("******** TRACE START ********~n"),
                    io:format("~p~n",[U_T]),
                    io:format("******** ^^^^ TRACE END ^^^^ ********~n"),
                    io:format("******** Population_Monitor:~p shut down with Reason:~p
OpTag:~p, while in OpMode:~p~n",[Population_Id,Reason,OpTag,OpMode]),
                    io:format("******** Tot Agents:~p Population Generation:~p
Tot_Evals:~p~n",[S#state.tot_agents,S#state.pop_gen,S#state.tot_evaluations]),
                    case S#state.benchmarker_pid of
                            undefined ->
```

```
                    ok;
                PId ->
                    PId ! {S#state.population_id,completed,U_T}
            end
    end.
```

With this done, and everything set up for the benchmarker to be able to spawn the population_monitor and store the experiment data if it wishes to do so, we now move forward to the next subsection and create this new benchmarker module.

13.4 Implementing the benchmarker

The benchmarker process will have three main functionalities:

1. To run the population_monitor N number of times, waiting for the population_monitor's trace after every run.
2. Create the experiment entry in the mnesia database, and keep updating its trace_acc as it itself accumulates the traces from the spawned population_monitors. The benchmarker should only do this if the backup_flag is set to true in the experiment record with which the benchmarker was started.
3. When the benchmarker has finished performing N number of evolutionary runs, and has accumulated N number of traces, it must print all the traces to console, calculate averages of the parameters between all the traces, and then finally write that data to file in the format which can be immediately graphed by GNUPlot.

In addition, because the benchmarker might be interrupted as it accumulates the traces, we want to build a function which can continue with the experiment when executed. Because each experiment will have its own unique Id, and because each experiment is stored to mnesia, this *continue* function should be executed with the *experiment id* parameter. When executed, it should read from the mnesia database all the needed information about the experiment, and then run the population_monitor the remaining number of times to complete the whole experiment.

In Listing-13.7 we implement the new benchmarker module. The comments after each function describe its functionality and purpose.

Listing-13.7 The implementation of the benchmarker module.

```
-module(benchmarker).
-compile(export_all).
-include("records.hrl").
%%% Benchmark Options %%%
-define(DIR,"benchmarks/").
```

```
-define(INIT_CONSTRAINTS,[#constraint{morphology=Morphology,
connection_architecture =CA, population_evo_alg_f=generational} || Morphology<-
[xor_mimic], CA<-[feedforward]]).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
start(Id)->
    PMP = #pmp{
        op_mode=gt,
        population_id=Id,
        survival_percentage=0.5,
        specie_size_limit=10,
        init_specie_size=10,
        polis_id = mathema,
        generation_limit = 100,
        evaluations_limit = 10000,
        fitness_goal = inf
    },
    E=#experiment{
        id = Id,
        backup_flag = true,
        pm_parameters=PMP,
        init_constraints = ?INIT_CONSTRAINTS,
        progress_flag=in_progress,
        run_index=1,
        tot_runs=10,
        started={date(),time()},
        interruptions=[]
    },
    genotype:write(E),
    register(benchmarker,spawn(benchmarker,prep,[E])).
%start/1 is called with the experiment id or name. It first assigns all the parameters to the pmp
and experiment records, and then writes the record to database (overwriting an existing one of
the same name, if present), and then finally spawns and registers the actual benchmarker pro-
cess.

continue(Id)->
    case genotype:dirty_read({experiment,Id}) of
        undefined ->
            io:format("Can't continue experiment:~p, not present in the database.~n",[Id]);
        E ->
            case E#experiment.progress_flag of
                completed ->
                    Trace_Acc = E#experiment.trace_acc,
                    io:format("Experiment:~p already completed:~p~n", [Id,
Trace_Acc]);
                in_progress ->
```

```
                        Interruptions = E#experiment.interruptions,
                        U_Interruptions = [now()|Interruptions],
                        U_E = E#experiment{
                                interruptions = U_Interruptions
                        },
                        genotype:write(U_E),
                        register(benchmarker,spawn(benchmarker,prep,[U_E]))
                end
    end.
%The continue/1 function spawns a benchmarker to continue a previously stopped experiment. If the experiment with the name/id of the Id parameter is already present in the database, and its progress_flag is set to in_progress, which means that the experiment has not yet completed and should continue running and accumulating new traces into its trace_acc list, then this function updates the experiment's interruptions list, and then spawns the benchmarker process using the experiment tuple as its parameter. The experiment record holds all the needed information to start the population_monitor, it contains a copy of the population monitor parameters, and the initial constraints used.

prep(E)->
    PMP = E#experiment.pm_parameters,
    U_PMP = PMP#pmp{benchmarker_pid=self()},
    Constraints = E#experiment.init_constraints,
    Population_Id = PMP#pmp.population_id,
    population_monitor:prep_PopState(U_PMP,Constraints),
    loop(E#experiment{pm_parameters=U_PMP},Population_Id).
%prep/1 function is run before the benchmarker process enters its main loop. This function extracts from the experiment all the needed information to run the population_monitor:prep_PopState/2 function and to start the population_monitor process with the right set of population monitor parameters and specie constraints.

loop(E,P_Id)->
    receive
            {P_Id,completed,Trace}->
                    U_TraceAcc = [Trace|E#experiment.trace_acc],
                    U_RunIndex = E#experiment.run_index+1,
                    case U_RunIndex >= E#experiment.tot_runs of
                            true ->
                                    U_E = E#experiment{
                                            trace_acc = U_TraceAcc,
                                            run_index = U_RunIndex,
                                            completed = {date(),time()},
                                            progress_flag = completed
                                    },
                                    genotype:write(U_E),
                                    report(U_E#experiment.id,"report");
```

```
                    false ->
                        U_E = E#experiment{
                            trace_acc = U_TraceAcc,
                            run_index = U_RunIndex
                        },
                        genotype:write(U_E),
                        PMP = E#experiment.pm_parameters,
                        Constraints = E#experiment.init_constraints,
                        population_monitor:prep_PopState(PMP,Constraints),
                        loop(U_E,P_Id)
                end;
            terminate ->
                ok
    end.
```

%loop/2 is the main benchmarker loop, which can only receive two types of messages, a trace from the population_monitor process, and a terminate signal. The benchmarker is set to run the experiment, and thus spawn the population_monitor process tot_runs number of times. After receiving the trace tuple from the population_monitor, it checks whether this was the last run or not. If it is not the last run, the benchmarker updates the experiment tuple, writes it to the database, and then spawns a new population_monitor by executing the population_monitor:prep_PopState/2 function. If it is the last run, then the function updates the experiment tuple, sets the progress_flag to completed, writes the updated experiment tuple to database, and runs the report function which calculates the averages and other statistical data, and produces the data for graphing, a file which can be used by the gnuplot program.

```
report(Experiment_Id,FileName)->
    E = genotype:dirty_read({experiment,Experiment_Id}),
    Traces = E#experiment.trace_acc,
    {ok, File} = file:open(?DIR++FileName++"_Trace_Acc", write),
    lists:foreach(fun(X) -> io:format(File, "~p.~n",[X]) end, Traces),
    file:close(File),
    io:format("******** Traces_Acc written to
file:~p~n",[?DIR++FileName++"_Trace_Acc"]),
    Graphs = prepare_Graphs(Traces),
    write_Graphs(Graphs,FileName++"_Graphs"),
    Eval_List = [T#trace.tot_evaluations||T<-Traces],
    io:format("Avg Evaluations:~p~n",[functions:avg(Eval_List),functions:std(Eval_List)]).
```

%report/2 is called with the id of the experiment to report upon, and the FileName to which to write the gnuplot formatted graphable data calculated from the given experiment. The function first extracts the experiment record from the database, then opens a file in the ?DIR directory to deposit the traces there, then calls the prepare_Graphs/1 function with the trace list from the experiment, and finally, with the data having now been prepared by the prepare_Graphs/1 function, the report function executes write_Graphs/2 to write the produced graphable data to the file FileName.

```
-record(graph,{morphology,avg_neurons=[],neurons_std=[],avg_fitness=[],fitness_std=[],
max_fitness=[], min_fitness=[],avg_diversity=[],diversity_std=[],evaluations=[],
evaluation_Index=[]}).
-record(avg,{avg_neurons=[],neurons_std=[],avg_fitness=[],fitness_std=[],max_fitness=[],
min_fitness=[], avg_diversity=[],diversity_std=[],evaluations=[]}).
%These two records contain the parameters specifically for the prepare_Graphs function. These
records are used to accumulate data needed to calculate averages and other statistical data from
the traces.

prepare_Graphs(Traces)->
    [T|_] = Traces,
    [Stats_List|_] = T#trace.stats,
    Morphologies = [S#stat.morphology || S<-Stats_List],
    Morphology_Graphs = [prep_Traces(Traces,Morphology,[])|| Morphology <-
Morphologies],
    [io:format("Graph:~p~n",[Graph])|| Graph<-Morphology_Graphs],
    Morphology_Graphs.
%prepare_Graphs/1 first checks a single trace in the Traces list to build a list of the morpholo-
gies present in the population (the number and types of which stays stable in our current im-
plementation throughout the evolutionary run), since the statistical data is built for each mor-
phology as its own specie. The function then prepares the graphable lists of data for each of the
morphologies in the trace. Finally, the function prints to screen the lists of values built from av-
eraging the traces. The data within the lists, like in the traces, is temporally sorted, composed
every 500 evaluations by default.

prep_Traces([T|Traces],Morphology,Acc)->
    Morphology_Trace=lists:flatten([[S||S<-Stats,S#stat.morphology==Morphology]||Stats<-
T#trace.stats]),
    prep_Traces(Traces,Morphology,[Morphology_Trace|Acc]);
prep_Traces([],Morphology,Acc)->
    Graph = avg_MorphologicalTraces(lists:reverse(Acc),[],[],[]),
    Graph#graph{morphology=Morphology}.
%prep_Traces/3 goes through every trace, and extracts from the stats list of those traces only
the stats associated with the morphology with which the function was called. Once the function
goes through every trace in the Traces list, and the morphologically specific trace data has been
extracted, the function calls avg_MorphologicalTraces/4 to construct a tuple similar to the
trace, but whose lists are composed of the average based values of all the morphology specific
traces, the average, std, max, min... of all the evolutionary runs in the experiment.

    avg_MorphologicalTraces([S_List|S_Lists],Acc1,Acc2,Acc3)->
            case S_List of
                    [S|STail] ->
                            avg_MorphologicalTraces(S_Lists,[STail|Acc1],[S|Acc2],Acc3);
                    [] ->
                            Graph = avg_statslists(Acc3,#graph{}),
```

```
                        Graph
        end;
    avg_MorphologicalTraces([],Acc1,Acc2,Acc3)->
        avg_MorphologicalTraces(lists:reverse(Acc1),[],[],[lists:reverse(Acc2)|Acc3]).
```

%avg_MorphologicalTraces/4 changes the dropped in S_lists from [Specie1_stats::[stat500, stat1000,...statN], Specie2_stats::[stat500,stat1000,...statN]...] to [[Spec1_Stat500, Spec2_Stat500... SpecN_Stat500], [Spec1_Stat1000, Spec2_Stat1000,... SpecN_Stat1000]...]. The trace accumulator contains a list of traces. A trace has a stats list, which is a list of lists of stat tuples. The stats list is a temporal list, since each stat list is taken every 500 evaluations, so the stats list traces-out the evolution of the population. Averages and other calculations need to be made for all experiments at the same temporal point, for example computing the average fitness between all experiments at the end of the first 500 evaluations, or at the end of the first 20000 evaluations... To do this, the function rebuilds the list from a list of separate temporal traces, to a list of lists where every such sublist contains the state of the specie (the stat) at that particular evaluation slot (at the end of 500, or 1000,...). Once this new list is built, the function calls avg_statslists/2, which calculates the various statistics of the list of lists.

```
    avg_statslists([S_List|S_Lists],Graph)->
        Avg = avg_stats(S_List,#avg{}),
        U_Graph = Graph#graph{
            avg_neurons = [Avg#avg.avg_neurons|Graph#graph.avg_neurons],
            neurons_std = [Avg#avg.neurons_std|Graph#graph.neurons_std],
            avg_fitness = [Avg#avg.avg_fitness|Graph#graph.avg_fitness],
            fitness_std = [Avg#avg.fitness_std|Graph#graph.fitness_std],
            max_fitness = [Avg#avg.max_fitness|Graph#graph.max_fitness],
            min_fitness = [Avg#avg.min_fitness|Graph#graph.min_fitness],
            evaluations = [Avg#avg.evaluations|Graph#graph.evaluations],
            avg_diversity = [Avg#avg.avg_diversity|Graph#graph.avg_diversity],
            diversity_std = [Avg#avg.diversity_std|Graph#graph.diversity_std]
        },
        avg_statslists(S_Lists,U_Graph);
    avg_statslists([],Graph)->
        Graph#graph{
            avg_neurons = lists:reverse(Graph#graph.avg_neurons),
            neurons_std = lists:reverse(Graph#graph.neurons_std),
            avg_fitness = lists:reverse(Graph#graph.avg_fitness),
            fitness_std = lists:reverse(Graph#graph.fitness_std),
            max_fitness = lists:reverse(Graph#graph.max_fitness),
            min_fitness = lists:reverse(Graph#graph.min_fitness),
            evaluations = lists:reverse(Graph#graph.evaluations),
            avg_diversity = lists:reverse(Graph#graph.avg_diversity),
            diversity_std = lists:reverse(Graph#graph.diversity_std)
        }.
```

%avg_statslists/2 calculates the averages and other statistics for every list in the S_lists, where each sublist is a list of stat tuples on which it executes the avg_stats/2 function, which returns

back a tuple with all the various parameters calculated from that list of stat tuples of that particular evaluations time slot.

```
avg_stats([S|STail],Avg)->
        U_Avg = Avg#avg{
                avg_neurons = [S#stat.avg_neurons|Avg#avg.avg_neurons],
                avg_fitness = [S#stat.avg_fitness|Avg#avg.avg_fitness],
                max_fitness = [S#stat.max_fitness|Avg#avg.max_fitness],
                min_fitness = [S#stat.min_fitness|Avg#avg.min_fitness],
                evaluations = [S#stat.evaluations|Avg#avg.evaluations],
                avg_diversity = [S#stat.avg_diversity|Avg#avg.avg_diversity]
        },
        avg_stats(STail,U_Avg);
avg_stats([],Avg)->
        Avg#avg{
                avg_neurons=functions:avg(Avg#avg.avg_neurons),
                neurons_std=functions:std(Avg#avg.avg_neurons),
                avg_fitness=functions:avg(Avg#avg.avg_fitness),
                fitness_std=functions:std(Avg#avg.avg_fitness),
                max_fitness=lists:max(Avg#avg.max_fitness),
                min_fitness=lists:min(Avg#avg.min_fitness),
                evaluations=functions:avg(Avg#avg.evaluations),
                avg_diversity=functions:avg(Avg#avg.avg_diversity),
                diversity_std=functions:std(Avg#avg.avg_diversity)
        }.
```

%avg_stats/2 function accepts a list of stat tuples as a parameter. First it extracts the various elements of that tuple. For every tuple in the list (each of the tuples belongs to a different evolutionary run) it puts the particular value of that tuple into its own list. Once all the values have been put into their own lists, the function uses the functions:avg/1 and functions:std/1 to calculate the averages and standard deviations as needed, to finally build the actual single tuple of said values (avg_neurons, neurons_std...). The case is slightly different for the max and min fitness values amongst all evolutionary runs, for which the function extracts the max amongst the maxs and the min amongst the mins, calculating the highest max and the lowest min achieved amongst all evolutionary runs. This can be further augmented to also simply calculate the avg of the max and min lists by changing the lists:min/1 and lists:max/1 to the function functions:avg/1.

```
write_Graphs([G|Graphs],Graph_Postfix)->
    Morphology = G#graph.morphology,
    U_G = G#graph{evaluation_Index=[500*Index || Index <-lists:seq(1,
length(G#graph.avg_fitness))]},
    {ok, File} = file:open(?DIR++"graph_"++atom_to_list(Morphology)++"_"
++Graph_Postfix, write),
    io:format(File,"#Avg Fitness Vs Evaluations, Morphology:~p~n",[Morphology]),
```

```
    lists:foreach(fun({X,Y,Std}) -> io:format(File, "~p ~p ~p~n",[X,Y,Std]) end,
lists:zip3(U_G#graph.evaluation_Index,U_G#graph.avg_fitness,U_G#graph.fitness_std)),
    io:format(File,"~n~n#Avg Neurons Vs Evaluations, Morphology:~p~n",[Morphology]),
    lists:foreach(fun({X,Y,Std}) -> io:format(File, "~p ~p ~p~n",[X,Y,Std]) end,
lists:zip3(U_G#graph.evaluation_Index,U_G#graph.avg_neurons,U_G#graph.neurons_std)),
    io:format(File,"~n~n#Avg Diversity Vs Evaluations, Morphology:~p~n",[Morphology]),
    lists:foreach(fun({X,Y,Std}) -> io:format(File, "~p ~p ~p~n",[X,Y,Std]) end,
lists:zip3(U_G#graph.evaluation_Index,U_G#graph.avg_diversity,U_G#graph.diversity_std)),
    io:format(File,"~n~n#Avg. Max Fitness Vs Evaluations, Morphology:~p~n",[Morphology]),
    lists:foreach(fun({X,Y}) -> io:format(File, "~p ~p~n",[X,Y]) end,
lists:zip(U_G#graph.evaluation_Index,U_G#graph.max_fitness)),
    io:format(File,"~n~n#Avg. Min Fitness Vs Evaluations, Morphology:~p~n",[Morphology]),
    lists:foreach(fun({X,Y}) -> io:format(File, "~p ~p~n",[X,Y]) end,
lists:zip(U_G#graph.evaluation_Index,U_G#graph.min_fitness)),
    io:format(File,"~n~n#Specie-Population Turnover Vs Evaluations, Morphology:~p~n",
[Morphology]),
    lists:foreach(fun({X,Y}) -> io:format(File, "~p ~p~n",[X,Y]) end,
lists:zip(U_G#graph.evaluation_Index,U_G#graph.evaluations)),
    file:close(File),
    write_Graphs(Graphs,Graph_Postfix);
write_Graphs([],_Graph_Postfix)->
    ok.
%write_Graphs/2 accepts a list of graph tuples, each of which was created for a particular specie/morphology within the experiment. Then for every graph, the function writes to file the various statistic results in the form readable by the gnuplot software. With the final result being a file which can be immediately used by the gnuplot to produce graphs of the various properties of the experiment.
```

With the benchmarker now implemented, we test it in the next subsection to ensure that all of its features are functional.

13.5 Compiling and Testing

Because we have created a new record, we now need to either add it to the mnesia database independently, or simply reset the whole thing (database), by executing the *polis:reset()* function. We now also need to test our new benchmarker system, and see whether it functions properly and does indeed save the data to the database, is able to continue the experiment after an interruption, and is able to produce a file which can be used by the gnuplot. Also, due to the following line in the benchmarker module: **-define(DIR,"benchmarks/")**, our benchmarker will be expecting for this folder to exist. Thus this folder must first be added, before we perform the following tests.

To test all these new features we will first recompile the code, and then reset the database. Afterwards, we will test our system in the following manner and order:

1. Set the benchmarker's *pmp* record to its current default, running the XOR mimicking experiment 10 times, to completion, using the generational evolutionary loop.
2. Examine the resulting console printout, to ensure basic structural validity, and that no crashes occurred.
3. Examine the two resulting files, the file that should have a list of traces, and the file which has data formatted in a gnuplot graphable format.
4. Plot the data in the graph based file, performing a basic sanity check on the resulting graph.
5. Again run the benchmarker, only this time, in the middle of the experiment execute: *Ctrl-C* to stop the interpreter midway, and then execute *'a'* to abort. This simulates the crashing of the machine in the middle of the experiment. We then re-enter the interpreter, and start up the polis to check whether the half finished experiment is present in the database. Once its presence is confirmed, we test *benchmarker:continue(Id)* by executing: *benchmarker:continue(test)*.
6. Finally, we examine the resulting console printout and the final experiment entry in the database, to ensure that the progress_flag is now set to: *completed*.

Because our implemented evolutionary loops (steady_state and generational) are independent of the evaluations accumulation, and thus the termination and the triggering of the benchmarker, we can simply perform these tests with the *generational* evolutionary loop, and not need to redo them with the steady_state evolutionary loop.

The default *pmp* and *experiment* records, and the ?INIT_CONSTRAINTS macro, are all set as follows:

```
-define(INIT_CONSTRAINTS,[#constraint{morphology=Morphology,connection_architecture
=CA, population_evo_alg_f=generational} || Morphology<-[xor_mimic],CA<-[feedforward]]).

#pmp{ op_mode=gt, population_id=test, survival_percentage=0.5, specie_size_limit=10,
init_specie_size=10, polis_id = mathema, generation_limit = 100, evaluations_limit = 10000,
fitness_goal = inf }

#experiment{ id = Id, backup_flag = true, pm_parameters=PMP, init_constraints
=?INIT_CONSTRAINTS, progress_flag=in_progress, run_index=1, tot_runs=10, start-
ed ={date(),time()}, interruptions=[] }
```

Having set everything to the intended values, we now (assuming that the new source has been compiled, and the new mnesia database has been created with all the appropriate tables by executing polis:reset()) run the *benchmarker:start(test)* function, as shown in Listing-13.8.

Listing-13.8 Running the benchmarker:start(test) function to test the benchmarker functionality.

```
2> benchmarker:start(test).
...
     [{stat,xor_mimic,7.544823116774118e-10,1.0,0.0,278.5367828868784,
          235.4058314015377,979.1905253086005,112.76113310465351,4,500,
          {1325,412119,825873}}]],
     10000,500}
******** ^^^^ TRACE END ^^^^ ********
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:10 Population Generation:36 Tot_Evals:10076
******** Traces_Acc written to file:"benchmarks/report_Trace_Acc"
Graph:{graph,xor_mimic,
[1.1345679012345677,1.3708193041526373,1.4792929292929293,...1.9777777777777774],
          [0.11516606301253175,0.3429379936199053,0.472713243338398,
...0.28588178511708023],
          [6376.044863498539,171964.06677104777,405553.7010466698,
...948483.9530134387],
          [13996.969949682387,305943.44537378295,421839.1376054512,
...46957.98926294873],
          [7595.914268861698,242099.32776384687,566599.7452288255,
...999402.6491394333],
[1736.703111779903,1157.4193567602842,227914.43647811364,...497519.90979294974],
          [5.111111111111111,6.444444444444445,...7.0],
          [0.7370277311900889,1.257078722109418,…2.1081851067789197],
          [500.0,500.0,500.0,500.0,500.0,500.0,444.44444444444446,...500.0],
          []}
```

It works! The console printout looks proper, a graph record, where each list is the average between all the experiments, with the averages calculated within the same evaluation frames. When we look into the *benchmark* folder, we see the presence of two files within: the *graph_xor_mimic_report_Graphs* file, and the *report_Trace_Acc* file. The report_Trace_Acc file contains a list of traces as expected, and shown in Listing-13.9.

Listing-13.9 The shortened contents of the report_Trace_Acc file.

```
{trace,[[{stat,xor_mimic,7.544235757558436e-10,2.0,0.0,999793.8069900939,
          20.85034690621442,999805.1547609345,999739.967822178,9,500,
          {1325,515312,752712}}],...
10000,500}.
{trace,[[{stat,xor_mimic,7.544235772700672e-10,2.0,0.0,999796.4301657086,
```

```
            3.35162014431123,999799.6097959183,999792.3483220651,8,500,
            {1325,515310,43590}}],...
10000,500}.
…
```

So far so good, the *report_Trace_Acc* contains all 10 traces. Another file, with the name graph_**xor_mimic**_*report_Graphs,* is also present in the benchmark folder. This file contains rows of values in the format we specifically created so that we can then use gnuplot to plot the resulting data. A sample of the formatted data within the file is shown in Listing-13.10.

Listing-13.10 The format of the graph_xor_mimic_report_Graphs file.

```
#Avg Fitness Vs Evaluations, Morphology:xor_mimic
500 6376.044863498539 13996.969949682387
1000 171964.06677104777 305943.44537378295
…
#Avg Neurons Vs Evaluations, Morphology:xor_mimic
500 1.1345679012345677 0.11516606301253175
1000 1.3708193041526373 0.3429379936199053
…
```

Again, after analyzing the graph, all the data seems to be in proper order. If we wish, we can use this file to create a plot using the gnuplot program. An example of such a plot is shown in Fig-13.1. Fig-13.1a and Fig-13.1b show the plots of Fitness (Avg, Max, and Min) vs. Evaluations, and Population Diversity vs. Evaluations, respectively. In Fig-13.1a we see that the average and max fitness quickly increases, and within the first 1000 evaluations they have already reached a very good score. The Min fitness within the graph is shown to always go up and down, as is expected, since every offspring might have a mutation which might make it ineffective. But even in that plot, we see that the minimum fitness also reaches high values, primarily because the mutations that break the system in some way, are mitigated by the tuning of the synaptic weights. In Fig-13.2b we see the diversity plotted against evaluations, with vertical error bars.

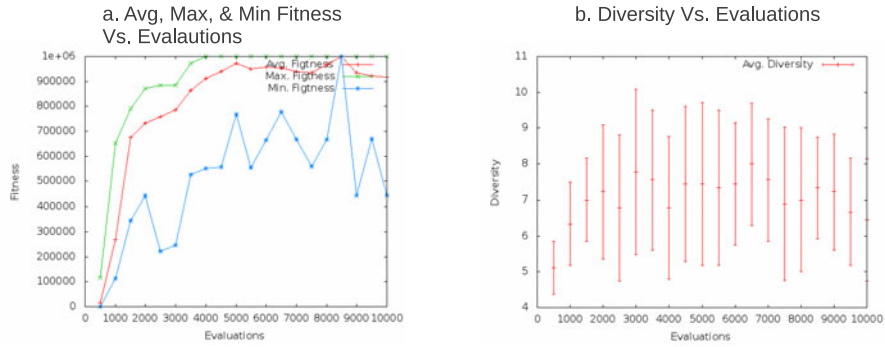

Fig. 13.1 The graphs produced with the data created by the benchmarker process, and plotted by the gnuplot program. Graph 'a' shows Fitness (Avg, Max, and Min) vs. Evaluations, and graph 'b' shows Diversity vs. Evaluations.

In the above figure we see that diversity never goes below 5 in a population of 10. A diversity of 4 is only present during the seed population, and primarily because there are only so many ways to create the minimalistic 1 neuron NN topology for this problem (through the use of different activation functions). The diversity in fact is increasing over time, not decreasing. The diversity reaches a stable value of 6-7, which means that 60%-70% of the population is different from one another, and the other 3-4 have similar topologies to those belonging to the 6-7 diverse topologies.

High population diversity is one of the important features of a memetic algorithm based TWEANN. In a system that we designed, it is simply not possible for diversity to shrink, because no matter which NN systems are fit or unfit, their offspring will have to be topologically different from them because they will pass through a topological mutation phase when created. As the size of the NN increases, so does the possible number of mutation operators applied to the clone during offspring creation, and thus the number of possible topological permutations, further increasing the number of mutants in the population, which results in an even higher diversity. As we increase the population size, again the result is greater diversity because now more agents can create offspring, and every one of those agents will produce a topological mutant, which will have a chance to be different from every other agent in the population and not just its parent.

Thus, a memetic algorithm based topology and weight evolving artificial neural network has a naturally emerging high diversity within its population, unlike the standard TWEANNs which usually converge very rapidly, and thus have a lower chance of solving the more complex problems. At the same time, the memetic TWEANN is also able to very rapidly solve problems it is applied to, and in my experience almost always faster than the standard TWEANN no matter the problem or simulation it is being used for. We will have a chance to test this bold claim when we benchmark our system against other TWEANNs in the following chapters.

With this done, we can now test the benchmarker's ability to continue a crashed or stopped experiment. You will most likely get a different result when testing on your machine, depending on when you stop the interpreter. On my machine, after having started the benchmarker, and then almost immediately stopping it by executing *Ctrl-C a*, and then re-entering the interpreter, my results were as follows when performing steps 5 and 6:

Listing-13.11 Crashing the benchmarker, and then attempting to continue by executing the benchmarker:continue(Id) function.

```
2> polis:start().
Parameters:{[],[]}
******** Polis: ##MATHEMA## is now online.
{ok,<0.35.0>}
2> benchmarker:start(test).
…
Ctrl-C
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
     (v)ersion (k)ill (D)b-tables (d)istribution
a
…
******** Polis: ##MATHEMA## is now online.
{ok,<0.34.0>}
2> mnesia:dirty_read({experiment,test}).
[{experiment,test,true,
          {pmp,gt,test,0.5,10,10,mathema,100,10000,inf,<0.143.0>},
          [{constraint,xor_mimic,feedforward, [tanh,cos,gaussian,absolute], [none],
[dot_product], [all],...],
          in_progress,
          [{trace,[[{stat,xor_mimic,7.544226409998199e-10,
                    2.0833333333333335,0.2763853991962833,833118.8760231837,
                    372581.937787711,999804.3485638215,0.34056136711788676,8,
                    500,
                    {1325,516955,41224}}],
...
                10000,500}],
          2,10,undefined,
          {{2012,1,2},{7,9,12}},
          undefined,[]}]
3>benchmarker:continue(test).
…
Graph:{graph,xor_mimic,...}
4>mnesia:dirty_read({experiment,test}).
…
   completed,
```

```
   …(TRACES)
   10,10,undefined,
start    {{2012,1,2},{7,9,12}},
end      {{2012,1,2},{7,14,51}},
         [{1325,517268,871875}]}]
```

It works! The benchmarker was first run and then abruptly stopped. After restarting the polis and checking the mnesia database, the experiment with the id *test* was present. Printing it to console showed, color coded in the above listing, that it contained the pmp record (green, and if you're reading the black & white printed version, it's the one starting with: "{pmp"), the constraints (blue, and starting with: "[{constraint"), and had a list of traces (red, and starting with: "[{trace"), 2 of which were present, out of the 10 the full experiment must contain. Finally, we also see the *in_progress* tag, which confirms that this experiment was stopped abruptly and is not yet finished. The function *benchmarker:continue(test)* was then executed, and the benchmarker ran to completion, printing the *Graph* tuple to console at the end. Finally, when rechecking the experiment entry in the database by executing *mnesia:dirty_read({experiment,test}),* we see that it contains 10 out of 10 evolutionary runs (traces), parameter *completed* is present, and we also see the start: **{{2012,1,2},{7,9,12}}** and end: **{{2012,1,2},{7,9,12}}** times respectively (which I marked with italicized "start" and "end" tags), are also present. The benchmarker works as expected, and we have completed testing it.

13.6 Summary

Every time an addition or extension is made to the neuroevolutionary system, it is important to see how it affects it as a whole. Is the neuroevolutionary system able to more effectively evolve agents? Is there high or low diversity? Does the neuroevolutionary approach taken converges too quickly, and is thus unable to inject enough diversity to overcome fitness walls present on the fitness landscape? Using a benchmarker helps us answer these questions.

We also created a new module called *benchmarker,* and a new table called *experiment,* within the database. The experiment table holds multiple complete experiment entries, each of which is composed of multiple traces, which are evolutionary runs applied to some problem. This allows for the experiment entry to be used to calculate the average performance of multiple runs of the same simulation/problem, thus giving us a general idea of how the system performs. We have created the benchmarker in such a way that it can run an experiment and save the traces to database after every successful run, such that in the case of a crash it can recover and continue with the experiment.

We are almost at the point where we can start adding new, much more advanced features. Features like plasticity, indirect encoding, crystallization... And though we can now perform benchmarks after adding such advanced features, we do not at this point have problems and simulations complex enough to test the new features on. Thus we first need to create this new set of more complex benchmarks and problems.

We need to create two types of new benchmarks. One standard neurocontroller benchmark, for which a recurrent and non recurrent solutions need to be evolved to solve it. This standard benchmark is called the pole balancing problem [3,4]. Another standard benchmark requires the NN based agent to learn as it interacts with the environment. We need such a benchmark to be able to tell whether the addition of neural plasticity to our evolved NN based systems improves them, and whether the added plasticity features work at all. The standard benchmark in this particular area is called the T-Maze navigation problem [5,6]. In the next chapter we will create both of these new problems, representing them as private scapes with which the evolving NN based agents can interact with.

13.7 References

[1] gnuplot: http://www.gnuplot.info/
[2] https://github.com/CorticalComputer/NeuroevolutionThroughErlang
[3] Gomez F, Miikkulainen R (1998). 2-D Pole Balancing with Recurrent Evolutionary Networks. In Proceedings of the International Conference on Artificial Neural Networks (Elsevier), pp. 2-7.
[4] Durr P, Mattiussi C, Floreano D (2006) Neuroevolution With Analog Genetic Encoding. Parallel Problem Solving from NaturePPSN iX, 671-680.
[5] Soltoggio A, Bullinaria JA, Mattiussi C, Durr P, Floreano D (2008) Evolutionary Advantages of Neuromodulated Plasticity in Dynamic, Reward-based Scenarios. Artificial Life 2, 569-576.
[6] Blynel J, Floreano D (2003) Exploring the T-maze: Evolving learning-like robot behaviors using CTRNNs. Applications of evolutionary computing 2611, 173-176.