# smarties

**Guido Novati**

**Jan 16, 2020**

# CONTENTS

**smarties is a Reinforcement Learning (RL) software designed with the following objectives**

- high-performance C++ implementations of [Remember and Forget for Experience Replay](https://arxiv.org/abs/1807.05827) and other deep RL learning algorithms including V-RACER, CMA, PPO, DQN, DPG, ACER, and NAF.
- the environment application determines at runtime the properties of the control problem to be solved. For example, the number of the agents in the environment, whether they are solving the same problem (and therefore they should all contribute to learning a common policy) or they are solving different problems (e.g. competing or collaborating). The properties of each agent's state and action spaces. Whether one or more agents are dealing with a partially-observable problem, which causes the learners to automatically use recurrent networks for function approximation. Whether the observation has to be preprocessed by convolutional layers and the properties thereof.
- the environment application is in control of the learning progress. More specifically, smarties supports applications whose API design is similar to that of OpenAI gym, where the environment is a self-contained function to be called to receive a new observation and advance the simulation in time. However, smarties also supports a more realistic API where the environment simulation dominates the structure of the application code. In this setting, it is the RL code that is called whenever new observations are available and require the agent to pick an action.
- the environment application determines the computational resources available to train and run simulations, with support for distributed (MPI) codebases.
- minimally intrusive plug-in API that can be inserted into C++, python and Fortran simulation software.

To cite this repository, reference the paper:

```
@inproceedings{novati2019a,
    title={Remember and Forget for Experience Replay},
    author={Novati, Guido and Koumoutsakos, Petros},
    booktitle={Proceedings of the 36\textsuperscript{th} International Conference on␣
↪Machine Learning},
    year={2019}
}
```

CHAPTER

ONE

# INSTALL

Regardless of OS, smarties relies on pybind11 for the Python interface. Therefore, to enable compilation of smarties with Python environment simulation codebases:

```
pip3 install pybind11 --user
```

Moreover, after the compilation steps for either Linux or Mac OS, add the path to the smarties library directory to your dynamic loader:

```
echo 'export SMARTIES_ROOT=/path/to/smarties/folder/' >> ~/.bash_profile
echo 'export PATH=${SMARTIES_ROOT}/bin:${PATH}' >> ~/.bash_profile
echo 'export LD_LIBRARY_PATH=${SMARTIES_ROOT}/lib:${LD_LIBRARY_PATH}' >> ~/.bash_
↪profile
echo 'export PYTHONPATH=${PYTHONPATH}:${SMARTIES_ROOT}/lib' >>  ~/.bash_profile
```

On Mac, `LD_LIBRARY_PATH` has to be replaced with `DYLD_LIBRARY_PATH`. The environment variable `SMARTIES_ROOT` is used to compile most of the applications in the 'apps' folder.

## 1.1 Linux

Smarties requires gcc version 6.1 or greater, a thread-safe (at least `MPI_THREAD_SERIALIZED`) implementation of MPI, and a serial BLAS implementation with CBLAS interface. Furthermore, in order to test on the benchmark problems, OpenAI gym or the DeepMind Control Suite with python>=3.5. MPI and OpenBLAS can be installed by running the `install_dependencies.sh` script.

```
git clone https://github.com/cselab/smarties.git
cd smarties
mkdir -p build
cd build
cmake ..
make -j
```

## 1.2 Mac OS

Installation on Mac OS is a bit more laborious due to to the LLVM compiler provided by Apple not supporting OpenMP threads. First, install the required dependencies as:

```
brew install llvm libomp open-mpi openblas
```

Now, we have to switch from Apple's LLVM compiler to the most recent LLVM compiler as default for the user's shell:

```
echo "alias cc='/usr/local/opt/llvm/bin/clang'" >> ~/.bash_profile
echo "alias gcc='/usr/local/opt/llvm/bin/clang'" >> ~/.bash_profile
echo "alias g++='/usr/local/opt/llvm/bin/clang++'" >> ~/.bash_profile
echo "alias c++='/usr/local/opt/llvm/bin/clang++'" >> ~/.bash_profile
echo "export PATH=/usr/local/opt/llvm/bin:\${PATH}" >> ~/.bash_profile
```

Then we are ready to get and install smarties:

```
git clone https://github.com/cselab/smarties.git
cd smarties/makefiles
make -j
```

CHAPTER

TWO

# ENVIRONMENT CODE SAMPLES

## 2.1 C++

The basic structure of a C++ based application for smarties is structured as:

```
#include "smarties.h"

inline void app_main(smarties::Communicator*const comm, int argc, char**argv)
{
  comm->setStateActionDims(state_dimensionality, action_dimensionality);
  Environment env;

  while(true) { //train loop
    env.reset(comm->getPRNG()); // prng with different seed on each process
    comm->sendInitState(env.getState()); //send initial state

    while (true) { //simulation loop
      std::vector<double> action = comm->recvAction();
      bool isTerminal = env.advance(action); //advance the simulation:

      if(isTerminal) { //tell smarties that this is a terminal state
        comm->sendTermState(env.getState(), env.getReward());
        break;
      } else  # normal state
        comm->sendState(env.getState(), env.getReward());
    }
  }
}

int main(int argc, char**argv)
{
  smarties::Engine e(argc, argv);
  if( e.parse() ) return 1;
  e.run( app_main );
  return 0;
}
```

For compilation, the following flags should be set in order for the compiler to find smarties:

```
LDFLAGS="-L${SMARTIES_ROOT}/lib -lsmarties"
CPPFLAGS="-I${SMARTIES_ROOT}/include"
```

## 2.2 Python

smarties uses pybind11 for seamless compatibility with python. The structure of the environment application is almost the same as the C++ version:

```
import smarties as rl

def app_main(comm):
  comm.setStateActionDims(state_dimensionality, action_dimensionality)
  env = Environment()

  while 1: #train loop
    env.reset() # (slightly) random initial conditions are best
    comm.sendInitState(env.getState())

    while 1: #simulation loop
      action = comm.recvAction()
      isTerminal = env.advance(action)

      if terminated:  # tell smarties that this is a terminal state
        comm.sendTermState(env.getState(), env.getReward())
        break
      else: # normal state
        comm.sendState(env.getState(), env.getReward())

if __name__ == '__main__':
  e = rl.Engine(sys.argv)
  if( e.parse() ): exit()
  e.run( app_main )
```

CHAPTER

THREE

# LAUNCHING

In many cases it is possible to launch an application compiled with smarties simple as, for example:

```
./exec [args...]
```

The script `smarties.py` is provided to allow greater flexibility, to ease passing options to smarties, and to help setting up MPI-based training processes. For example, to have multiple processes running the environment (distributed data-collection) or multiple processes hosting the RL algorithms (distributed SGD).

With the `bin` directory added to the shell `PATH`, the description of the setup options are printed out by typing:

```
smarties.py --help
```

The script takes 2 (optional) positional arguments, for example:

```
smarties.py cart_pole_py VRACER.json
```

In this case, smarties will train with the V-RACER algorithm, and hyper-parameters defined in the `VRACER.json` file found in the `SMARTIES_ROOT/settings` directory on the application `cart_pole_py` found in the `SMARTIES_ROOT/apps` folder. All output files will be saved in the current directory. If no arguments are provided, the script will look for an executable (named `exec` or `exec.py` in the current directory or whatever specified with the `--execname exec` option) and will use default hyper-parameters.

Most useful options:

- `--gym` to tell smarties to run OpenAI gym applications (eg. `smarties.py Walker2d-v2 --gym`)
- `--atari` to tell smarties to run OpenAI gym Atari applications. For example, `smarties.py Pong --atari` will run the `PongNoFrameskip-v4` environment with DQN-like preprocessing conv2d layers as specified by `apps/OpenAI_gym_atari/exec.py`.
- `--dmc` to tell smarties to run DeepMind Control Suite applications. For example, `smarties.py "acrobot swingup" --dmc` will run the `acrobot` environment with task `swingup`.
- `--runname RUNNAME` will execute the training run from folder `RUNNAME` and create all output and setup files therein. The path of the folder is by default `SMARTIES_ROOT/runs/RUNNAME`, but may be modified for example as `--runprefix ./`, which will create `RUNNAME` in the current directory.
- `--nEvalSeqs N` tells smarties that it should evaluate and not modify an already trained policy for `N` sequences (the smarties-generated restart files should be already located in the run directory or at path `--restart /path/to/restart/`).
- `--args "arg1 arg2 .. ` in order to pass line arguments to the application.
- `--nEnvironments N` will spawn `N` processes running environment simulations. If the environment requires (or benefits from) one or more dedicated MPI ranks (recommended for clusters and expensive simulations) this can be set with `--mpiProcsPerEnv M`. In this case, 1+N*M MPI processes will run the training: one

learner and N teams of M processes to handle the N simulations. If the network update needs to be parallelized (distributed SGD), use the option `--nLearners K`.

Note for evaluating trained policies. For safety, use the option `--restart` or copy all the `agent_[...].raw` files onto a new folder in order to not overwrite any file of the training directory. Make sure the policy is read correctly (eg. if code was compiled with different features or run with different algorithms) comparing the `restarted_[...]` files and the originals (e.g. `diff /path/eval/run/restarted_agent_00_net_weights.raw /path/train/run/agent_00_net_weights.raw`).

## 3.1 Hands-on examples

The `apps` folder contains a number of examples showing the various use-cases of smarties. Each folder contains the files required to define and run a different application. While it is generally possible to run each case as `./exec` or `./exec.py`, smarties will create a number of log files, simulation folders and restart files. Therefore it is recommended to manually create a run directory or use the launch scripts contained in the `launch` directory.

The applications that are already included are:

- **`apps/cart_pole_cpp`: simple C++ example of a cart-pole balancing problem.** Assuming, all steps in the Install section were successful, compile the application: `cd apps/cart_pole_cpp && make`. As described above, running this application can be done as:
  - From the `cart_pole_cpp` directory, `mkdir test && ../cart_pole`. Here we create a new directoy, where all logging, saving, and postprocessing files will be created by smarties, and run the application directly. Because we do not use MPI, smarties will fork two processes, one running the environment (described by the application cart_pole) and one will run the training.
  - From the `cart_pole_cpp` directory, `smarties.py -r test`. Here we rely on the helper script to create the directory `test` which by default will be placed in `${SMARTIES_ROOT}\runs\`.
  - From any directory, `smarties.py cart_pole_cpp -r test` or `smarties.py apps/cart_pole_cpp -r test`. Refer to the section above and `smarties.py --help` for more customization options.
- `apps/cart_pole_py`: simple python example of a cart-pole balancing problem. Can be run similarly to the C++ code: `mkdir test && ../cart_pole.py` or `mkdir test && python3 ../cart_pole.py`.
- `apps/cart_pole_f90`: simple fortran example of a cart-pole balancing problem
- `apps/cart_pole_many`: example of two cart-poles that define different decision processes: one performs the opposite of the action sent by smarties and the other hides some of the state variables from the learner (partially observable) and tehrefore requires recurrent networks.
- `apps/cart_pole_distribEnv`: example of a distributed environment which requires MPI. The application requests M ranks to run each simulation. If the executable is ran as `mpirun -n N exec`, (N-1)/M teams of processes will be created, each with its own MPI communicator. Each simulation process contains one or more agents.
- `apps/cart_pole_distribAgent`: example of a problem where the agent themselves are distributed. Meaning that the agents exist across the team of processes that run a simulation and get the same action to perform. For example flow actuation problems where there is only one control variable (eg. some inflow parameter), but the entire simulation requires multiple CPUs to run.
- `apps/predator_prey`: example of agents competing.
- `apps/glider`: example of an ODE-based control problem that requires precise controls, used for the paper [Deep-Reinforcement-Learning for Gliding and Perching Bodies](https://arxiv.org/abs/1807.03671)

- `apps/func_maximization/`: example of function fitting and maximization, most naturally approached with CMA.
- `apps/OpenAI_gym`: code to run most gym application, including the MuJoCo based robotic benchmarks shown in [Remember and Forget for Experience Replay](https://arxiv.org/abs/1807.05827)
- `apps/OpenAI_gym_atari`: code to run the Atari games, which automatically creates the required convolutional pre-processing
- `apps/Deepmind_control`: code to run the Deepmind Control Suite control problems
- `apps/CUP2D_2fish`: and similarly named applications require CubismUP 2D.

## 3.2 Examples of solved problems

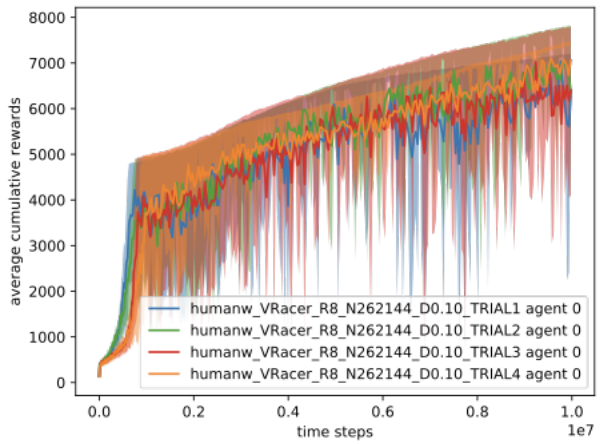
- The first two visualizations are from G. Novati and P. Koumoutsakos, "Remember and forget for experience replay," in Proceedings of the 36th international conference on machine learning, 2019.
- The fifth is from S. Verma, G. Novati, and P. Koumoutsakos, "Efficient collective swimming by harnessing vortices through deep reinforcement learning," Proceedings of the national academy of sciences, p. 201800923, 2018.
- The fourth is from G. Novati, S. Verma, D. Alexeev, D. Rossinelli, W. M. van Rees, and P. Koumoutsakos, "Synchronisation through learning for two self-propelled swimmers," Bioinspiration & biomimetics, vol. 12, iss. 3, p. 36001, 2017.
- Se also G. Novati, L. Mahadevan, and P. Koumoutsakos, "Controlled gliding and perching through deep-reinforcement-learning," Physical review fluids, vol. 4, iss. 9, 2019 for an introduction to using deep RL to obtain optimal control policies in fluid mechanics problems.

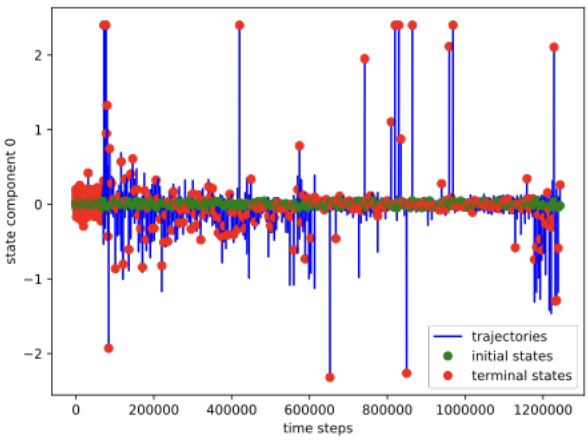
CHAPTER

FOUR

# OUTPUTS AND POSTPROCESSING

- Running the script will produce the following outputs on screen (also backed up into the files `agent_%02d_stats.txt`). According to applicability, these are either statistics computed over the past 1000 steps or are the most recent values:
  - `ID`: Learner identifier. If a single environment contains multiple agents, and if each agent requires a different policy, then we distinguish outputs pertinent to each agent with this ID integer.
  - `#/1e3`: Counter of gradient steps divided by 1000
  - `avgR | stdr | DKL`: Average **cumulative** reward among stored episodes, standard dev of the distribution of **instantaneous** rewards, and average Kullback Leibler divergence of experiences in the Memory Buffer w.r.t. current policy.
  - `nEp | nObs | totEp | totObs | oldEp | nFarP`: Number of episodes and observations in the Replay Memory. Total ep/obs since beginning of training passing through the buffer. Time stamp of the oldest episode (more precisely, of the last observation of the episode) that is currently in the buffer. Number of far-policy samples in the buffer.
  - `net` and/or `policy` and/or `critic` and/or `input` and/or other: L2 norm of the weights of the corresponding network approximator.
  - `RMSE | avgQ | stdQ | minQ | maxQ`: RMSE of Q (or V) approximator, its average value, standard deviation, min and max.
  - (if algorithm employs parameterized policy) `polG | penG | proj` Average norm of the policy gradient and that of the penalization gradient (if applicable). Third is the average projection of the policy gradient over the penalty one. I.e. the average value of `proj = polG \cdot penG / sqrt(penG \cdot penG)`. `proj` should generally be negative: current policy should be moved away from past behavior in the direction of pol grad.
  - (extra outputs depending on algorithms) In RACER/DPG: `beta` is the weight between penalty and policy gradients. `avgW` is the average value of the off policy importance weight `pi/mu`. `dAdv` is the average change of the value of the Retrace estimator for a state-action pair between two consecutive times the pair was sampled for learning. In PPO: `beta` is the coefficient of the penalty gradient. `DKL` is the average Kullback Leibler of the 'proximally' on-policy samples used to compute updates. `avgW` is the average value of `pi/mu`. `DKLt` is the target value of Kullback Leibler if algorithm is trying to learn a value for it.




- The file `agent_%02d_rank%02d_cumulative_rewards.dat` contains the all-important cumulative rewards. It is stored as text-columns specifying: gradient count, time step count, agent id, episode length (in time steps), sum of rewards over the episode. The first two values are recorded when the last observation of the episode has been recorded. Can be plotted with the script `smarties_plot_rew.py` script (eg. the figure on the left above). `smarties_plot_rew.py` accepts a list of run directories and optional arguments explained by `marties_plot_rew.py --help`.
- If data logging was not disabled (option `--disableDataLogging` for `smarties.py`), a complete log of all state/action/rewards/policies will be stored in binary files named `agent_02d_rank%02d_obs.raw`. These can be plotted by the script `smarties_plot_obs.py` (eg. the figure on the right above). The help message is straightforward.
- The files named `agent_%02d_${network_name}_${SPEC}.raw` contain back-ups of network weights (`weights`), Adam's moments estimates (`1stMom` and `2ndMom`) and target weights (`tgt_weights`) at regularly spaced time stamps. Some insight into the shape of the weight vector can be obtained by plotting with the script `smarties_plot_weights.py`. The files ending in `scaling.raw` contain the values used to rescale the states and rewards. Specifically, one after the other, 3 arrays of size `d_S` of the state-values means, 1/stdev, and stdev, followed by one value corresponding to 1/stdev of the rewards.
- The files `agent_%02d_${network_name}_grads.raw` record the statistics (mean, standard deviation) of the gradients received by each network output. Can be plotted with `smarties_plot_grads.py`.
- Various files ending in `.log`. These record the state of smarties on startup. They include: `gitdiff.log` records the changes wrt the last commit, `gitlog.log` records the last commits, `out.log` is a copy of the screen output, and `problem_size.log` records state/action sizes used by other scripts.

CHAPTER

FIVE

# API FUNCTIONS

Here are reported all the functions available through the *Communicator* passed by smarties to the environment *app_main* function (see *Environment code samples*). The main difference between using these functions with Python, as opposed to C++, is that Python lists or numpy arrays are used in place of *std::vector<double>*.

Use *python3 -c 'import smarties as rl; help(rl)'* when in doubt.

## 5.1 Core RL loop

These function are all thread-safe (not in Python, obviously) as long as different threads use different agentIDs.

```
void sendInitState(const std::vector<double>& state, const int agentID = 0)
```

Send the first state of a new episode for agent # 'agentID'. Because no action has been done yet there is no reward.

```
void sendState(const std::vector<double>& state, const double reward, const int␣
↪agentID = 0)
```

Send normal state and reward for agent # 'agentID'.

```
void sendTermState(const std::vector<double>& state, const double reward, const int␣
↪agentID = 0)
```

Send terminal state and reward for agent # 'agentID'. Note: V(s_terminal) = 0 because episode cannot continue. For example, agent succeeded in task, or is incapacitated, or time ran out on a time-constrained task.

```
void sendLastState(const std::vector<double>& state, const double reward, const int␣
↪agentID = 0)
```

Send last state and reward of the episode for agent # 'agentID'. Note: This corresponds to V(s_last) != 0 and it implies that it would be possible to continue the episode with this policy. In other words, timeout is not caused by the agent's policy. For example, when a robot is learning to perform a repetitive task (e.g. walk) and there is some arbitrary time horizon (e.g. in OpenAI gym). Or in an environment where multiple cars are being driven by RL and which requires a full reset after each collision between cars. Two cars might crash and reach their terminal state. In this case, the cars not involved in the collision would be in a 'last state', because their policy was not cause for termination.

```
std::vector<double>& recvAction(const int agentID = 0)
```

Get the action for agent # 'agentID' selected by the RL algorithm according to the previously sent state (either initial or normal). Cannot be called after a last or terminal state.

## 5.2 Problem specification

These functions have to be used before calling 'sendInitState' for the first time.

```
void setNumAgents(int nAgents)
```

Set number of agents in the environment.

```
void setStateActionDims(const int dimState, const int dimAct, const int agentID = 0)
```

Set dimensionality of state and action for agent # 'agentID'.

```
void setActionScales(const std::vector<double> upper, const std::vector<double> lower,
↪ const bool bound, const int agentID = 0)
```

Set lower and upper scale of the actions for agent # 'agentID'. Boolean arg specifies if actions are bounded between given values. Implies continuous action spaces.

```
void setActionScales(const std::vector<double> upper, const std::vector<double> lower,
↪ std::vector<bool> bound, const int agentID = 0)
```

Set lower and upper scale of the actions for agent # 'agentID'. Boolean vector specifies if actions components are bounded between gien values. Implies continuous action spaces.

```
void setActionOptions(const int options, const int agentID = 0)
```

Set number of discrete control options for agent # 'agentID'. Implies discrete action spaces.

```
void setActionOptions(const std::vector<int> options, const int agentID = 0)
```

Set number of discrete control options for agent # 'agentID' in case of multi-dimensional options vectors (e.g. choose to turn left/right and shoot/dontshoot). Reduntant because can be directly mapped onto the previous function. Implies discrete action spaces.

```
void setStateObservable(const std::vector<bool> observable, const int agentID = 0)
```

For each state variable, set whether observed by agent # 'agentID'. Allows hiding state components from agent (will not be included in policy/value networks) or passing auxilliary observables to smarties such that they will be logged to file.

```
void setStateScales(const std::vector<double> upper, const std::vector<double> lower,␣
↪const int agentID = 0)
```

Set upper & lower scaling values for the state of agent # 'agentID'.

```
void setIsPartiallyObservable(const int agentID = 0)
```

Specify that the decision process of agent # 'agentID' is non-Markovian and therefore smarties will use RNN.

## 5.3 Advanced problem specification

```
void envHasDistributedAgents()
```

Returns true if smarties is training, false if evaluating a policy.

```
void agentsDefineDifferentMDP()
```

Specify that each agent defines a different MPD (state/action/rew). This means that smarties will train a separate policy for each agent in the environment. All problem specification settings submitted before calling this function will be shared among all agents.

```
void disableDataTrackingForAgents(int agentStart, int agentEnd)
```

Set agents whose experiences should not be used as training data.

```
void agentsShareExplorationNoise(const int agentID = 0)
```

```
void setPreprocessingConv2d(const int input_width, const int input_height, const int␣
↪input_features, const int kernels_num, const int filters_size, const int stride,␣
↪const int agentID = 0)
```

Request a convolutional layer in preprocessing of state for agent # 'agentID'. This function can be called multiple times to add multiple conv2d layers, but sizes (widths, heights, filters) must be consistent otherwise it will trigger an abort.

```
void setNumAppendedPastObservations(const int n_appended, const int agentID = 0)
```

Specify that the state of agent # 'agentID' should be composed with the current observation along with n_appended past ones. Like it was done in the Atari Nature paper to avoid using RNN.

```
void finalizeProblemDescription()
```

Signals that problem formulation will not be changed further. This function is otherwise called automatically by smarties before the first 'sendInitState'. It is only required that the user explicitly calls this function before starting the training loop for multi-threaded environments. In this case, multiple threads might attempt to call this function during their first 'sendInitState', which is not thread-safe.

## 5.4 Utility

These functions can be called at any time.

```
std::mt19937& getPRNG()
```

Passes a random number generator. C++ only.

```
Real getUniformRandom(const Real begin = 0, const Real end = 1)
```

Returns an uniformly distributed real number.

```
Real getNormalRandom(const Real mean = 0, const Real stdev = 1)
```

Returns a normally distributed real number.

```
bool isTraining()
```

Returns true if smarties is training, false if evaluating a policy.

```
bool terminateTraining()
```

Returns true if smarties is requesting application to exit. If application does not return after smarties requests an exit smarties will trigger an abort (inelegant exit).

## 5.5 Function optimization interface

```
const std::vector<double>& getOptimizationParameters(int agentID = 0)
```

```
void setOptimizationEvaluation(const Real R, const int agentID = 0)
```