Arcade Learning Environment Technical Manual (v.0.6.0)

Marlos C. Machado, Matthew Hausknecht, Marc G. Bellemare

October 14, 2015

Contents

1 Overview

This document is roughly divided into three parts. The first part describes how to install the Arcade Learning Environment (ALE). The second part describes the various ALE interfaces currently available:

1. **Shared Library interface** (C++ only): Loads ALE as a shared library (Section 3).
2. **CTypes interface** (Python only): A fast Python interface to ALE, provided as a Python package (Section 4).
3. **FIFO interface** (all languages): Communicates with ALE through a text interface (Section 5).
4. **RL-Glue interface** (C/C++, Java, Python, Matlab, Lisp, Go): Communicates with ALE via RL-Glue (Section 6).

The final part of this document discusses the different features of ALE, including the action stochasticity parameter and video recording.

ALE also provides an example Java agent, not discussed in this manual, which uses the FIFO interface. This agent includes code for human input as well as a simple SARSA implementation. Details on the Java agent, including installation instructions, may be found under `doc/java-agent`.

2 Installing

2.1 Requirements

The basic requirements to build and run ALE are:

- `CMake or make`
- A C++ compiler

Two custom makefiles are provided, `makefile.mac` (Mac OS X) and `makefile.unix` (Linux). However, beginning with version 0.5.0, we highly recommend the use of CMake rather than the custom makefiles.

Additionally, the following options are not activated by default and require additional packages:

- SDL display and audio support
- RL-Glue support

2.2 Installation/Compilation

One first has to install ALE requirements before compiling ALE itself. We assume that a suitable C++ compiler (one which includes `make`) is already available to the user. Installing CMake (and if desired, SDL) is straightforward, and can be done through package managers:

Mac OS X (using Homebrew[1]):

```
> brew install cmake
> brew install sdl
```

Linux (using apt, e.g. Ubuntu):

```
> sudo apt-get install cmake
> sudo apt-get install libsdl1.2-dev
```

For instructions on installing RL-Glue, see Section 6.

We are going to assume ALE was extracted to `ale_0_5`. Then, compiling it with CMake is very simple (on both systems):

```
> cd ale_0_5
ale_0_5> cmake -DUSE_SDL=ON -DUSE_RLGLUE=OFF -DBUILD_EXAMPLES=ON .
ale_0_5> make -j 4
```

This compiles the code with SDL but not RL-Glue, and builds the example C++ agents described in this document. The first two options are disabled by default, while the third is enabled; default options may be omitted from the command line. Note that SDL and CMake sometimes don't play well together; please refer to Troubleshooting (Section 9) if compilation fails, e.g. because of a missing `SDL.h`.

3 C++ Agent: Shared Library Interface

The shared library interface is the simplest way to implement a C++ agent for ALE. This interface allows agents to directly access ALE via a class called `ALEInterface`, defined in `ale_interface.hpp`. Example code detailing a simple random agent is provided under `doc/examples/sharedLibraryInterfaceExample.cpp`.

To implement an agent, the first step is to include the library `ale_interface.hpp`, either via the relative path `#include "path/from/your/code/ale_interface.hpp"`, or as a standard header: `#include <ale_interface.hpp>`. If the later is chosen, remember to add the proper path using the flag `-I` when compiling the code.

To instantiate the Arcade Learning Environment it is enough to write:

[1] http://brew.sh

```
ALEInterface ale;
```

Once the environment is initialized, it is now possible to set its arguments. This is done with the functions `setBool(), setInt(), setFloat()`. The complete list of flags is available in Section 11. Just as an example, to set the environment's seed we write:

```
ale.setInt("random_seed", 123);
```

Finally, after setting the desired environment parameters we now load the game ROM by providing its filename to the `loadROM` method:

```
ale.loadROM("asterix.bin");
```

There are two different action sets provided by ALE: the "legal" set and the "minimal" set. Save for a few rare exceptions, the legal action set consists of all 18 actions for all games, including duplicates and actions with no effect. On the other hand, the minimal action set for a game contains only the actions that have some effect on that game. The `getLegalActionSet` and `getMinimalActionSet` methods provide the desired action sets:

```
ActionVect legal_actions = ale.getLegalActionSet();
```

Taking an action is done by calling the function `act()` passing an object of `Action` as a parameter:

```
Action a = legal_actions[rand() % legal_actions.size()];
float reward = ale.act(a);
```

Finally, one can check whether the episode has terminated using the function `ale.game_over()`. With these functions one can already implement a very simple agent that plays randomly for one episode:

```
#include <ale_interface.hpp>

using namespace std;

int main(int argc, char** argv) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " rom_file" << std::endl;
        return 1;
    }

    ALEInterface ale;
    ale.setInt("random_seed", 123);
```

```
    ale.loadROM(argv[1]);

    ActionVect legal_actions = ale.getLegalActionSet();

    float totalReward = 0.0;
    while (!ale.game_over()) {
        Action a = legal_actions[rand() % legal_actions.size()];
        float reward = ale.act(a);
            totalReward += reward;
        }
        std::cout << "The episode ended with score: " << totalReward
            << std::endl;
    }
    return 0;
}
```

Compiling with the shared library is done by appending the `-lale` flag. See the

`doc/examples/sharedLibraryExample.cpp`

example for more details, including compilation, as well as Section 9 if errors arise. A complete list of functions available in the class `ALEInterface` is given in Section 10.

4 Python Agent: CTypes Interface

To use the Python interface it is necessary to install it after ALE was compiled. With root/superuser access:

```
pip install .
```

without root/superuser access:

```
pip install --user .
```

This will install the package `ale_python_interface` which can be imported as usual. Example code is available under

`doc/examples/python_example.py`.

Aside from a few minor differences, the Python interface mirrors the C++ interface. For example, the following implements a random agent:

```
import sys
from random import randrange
from ale_python_interface import ALEInterface
```

```
if len(sys.argv) < 2:
  print 'Usage:', sys.argv[0], 'rom_file'
  sys.exit()

ale = ALEInterface()
ale.setInt('random_seed', 123)
ale.loadROM(sys.argv[1])

# Get the list of legal actions
legal_actions = ale.getLegalActionSet()

total_reward = 0
while not ale.game_over():
  a = legal_actions[randrange(len(legal_actions))]
  reward = ale.act(a);
  total_reward += reward
print 'Episode ended with score:', total_reward
```

5 FIFO Interface

The FIFO interface is text-based and allows the possibility of run-length encoding the screen. This section documents the actual protocol used; sample code implementing this protocol in Java is also included in this release.

After preliminary handshaking, the FIFO interface enters a loop in which ALE sends information about the current time step and the agent responds with both players' actions (in general agents will only control the first player). The loop is exited when one of a number of termination conditions occurs.

5.1 Handshaking

ALE first sends the width and height of its screen matrix as a hyphen-separated string:

`www-hhh\n`

where `www` and `hhh` are both integers. The agent then responds with a comma-separated string:

`s,r,k,R\n`

where `s`, `r`, `R` are 1 or 0 to indicate that ALE should or should not send, at every time step, screen, RAM and episode-related information (see below for details). The third argument, `k`, is deprecated and currently ignored.

5.2 Main Loop – ALE

After handshaking, ALE will then loop until one of the termination conditions occurs; these conditions are described below in Section 5.4. If terminating, ALE sends

```
DIE\n
```

Otherwise, ALE sends

```
<RAM_string><screen_string><episode_string>\n
```

Where each of the three strings is either the empty string (if the agent did not request this particular piece of information), or the relevant data terminated by a colon.

5.2.1 `RAM_string`

The RAM string is 128 2-digit hexadecimal numbers, with the i^{th} pair denoting the i^{th} byte of RAM; in total this string is 256 characters long, not including the terminating ':'.

5.2.2 `screen_string`

In "full" mode, the screen string is `www` $\times$ `hhh` 2-digit hexadecimal numbers, each representing a pixel. Pixels are sent row by row, with `www` characters for each row. In total this string is $2 \times$ `www` $\times$ `hhh` characters long.

In run-length encoding mode, the screen string consists of a variable number of (colour, length) pairs denoting a run-length encoding of the screen, also row by row. Both colour and length are described using 2-digit hexadecimal numbers. Each pair indicates that the next 'length' pixels take on the given colour; run length is thus limited to 255. Runs may wrap around onto the next row. The encoding terminates when the last pixel (i.e. the bottom-right pixel) is encoded. The length of this string is 4 characters per (colour, length) pair, and varies depending on the screen.

In either case, the screen string is terminated by a colon.

5.2.3 `episode_string`

The episode string contains two comma-separated integers indicating episode termination (1 for termination, 0 otherwise) and the most recent reward. It is also colon-terminated.

5.2.4 Example

Assuming that the agent requested screen, RAM and episode-related information, a string sent by ALE might look like:

```
000100...A401B2:3C3C3C3C00003C3C3C...4F4F0000:0,1:\n
^ 2x128 characters   ^ 2x160x210 characters    ^ongoing episode, reward of 1
```

5.3 Main Loop – Agent

After receiving a string from ALE, the agent should now send the actions of player A and player B. These are sent as a pair of comma-separated integers on a single line, e.g.:

```
2,18\n
```

where the first integer is player A's action (here, FIRE) and the second integer, player B's action (here, NOOP). Emulator control (reset, save/load state) is also handled by sending a special action value as player A's action. See Section 7.1 for the list of available actions.

5.4 Termination

ALE will terminate (and potentially send a `DIE` message to the agent) whe one of the following conditions occur:

- `stdin` is closed, indicating that the agent is no longer sending data, or
- the maximum number of frames (user-specified, with no maximum by default) has been reached.

ALE will send an end-of-episode signal when one the following is true:

- The maximum number of frames for this episode (user-specified, with no maximum by default) has been reached, or
- the game has ended, usually when player A loses their last life.

6 RL-Glue Interface

The RL-Glue interface implements the RL-Glue 3.0 protocol. It requires the user to first install the RL-Glue core. Additionally, the example agent and environment require the RL-Glue C/C++ codec. Both of these can be found on the RL-Glue web site[2].

In order to use the RL-Glue interface, ALE must be compiled with RL-Glue support. This is achieved by invoking CMake with `-DUSE_RLGLUE=ON` (see Section 2) or, if using custom makefiles, setting `USE_RLGLUE := 1`.

Specifying the command-line argument `-game_controller rlglue` is sufficient to start ALE in RL-Glue mode. It will then communicate with the RL-Glue core like a regular RL-Glue environment.

[2] http://glue.rl-community.org

6.1 Sample Agent and Experiment

Recall that RL-Glue is a combination of four processes: a core, an experiment, an agent, and an environment. The core is provided by the RL-Glue library itself, and ALE is the environment here. An example experiment and agent are provided in

```
doc/examples/RLGlueExperiment.c
doc/examples/RLGlueAgent.c
```

Assuming you installed ALE under `ale_0_5`, the RL-Glue agent and experiment can be compiled with the following command:

```
ale_0_5/doc/examples> make -f Makefile.rlglue
```

You will then need to start the following processes to run the sample RL-Glue agent in ALE:

```
ale_0_5> rl_glue
ale_0_5> doc/examples/RLGlueAgent
ale_0_5> doc/examples/RLGlueExperiment
ale_0_5> ./ale -game_controller rlglue
```

Additional options (such as those described below) can also be passed as command-line arguments. Please refer to the RL-Glue documentation for more details.

6.2 Actions and Observations

The action space consists of both Player A and Player B's actions (see Section 7.1 for details). In general, Player B's action may safely be set to noop (18) but it should be left out altogether if the `restricted_action_set` option is set to true.

The observation space depends on whether the `send_rgb` option is enabled. When enabled, the observation space consists of 100,928 integers: first the 128 bytes of RAM (taking values in 0–255), followed by 100,800 bytes describing the screen. Each pixel is described by three bytes, taking values in 0–255, specifying the pixel's red, green and blue components in that order. The screen is provided in row-order, beginning with the 160 pixels that compose the first row.

If `send_rgb` is disabled (this is the default), the observation space consists of 33,728 integers: first the 128 bytes of RAM, then the 33,600 screen pixels (in NTSC format; values in 0–127). These pixels are also provided in row order.

7 Environment Specifications

This section provides additional information regarding the environment implemented in ALE.

7.1 Available Actions

The following regular actions are defined in `common/Constants.h` and interpreted by ALE:

noop (0)	fire (1)	up (2)	right (3)	left (4)
down (5)	up-right (6)	up-left (7)	down-right (8)	down-left (9)
up-fire (10)	right-fire (11)	left-fire (12)	down-fire (13)	up-right-fire (14)
up-left-fire (15)	down-right-fire (16)	down-left-fire (17)	reset* (40)	

Note that the `reset` (40) action toggles the Atari 2600 switch, rather than reset the environment, and as such is ignored by most interfaces. The shared library, CTypes, and fifo interfaces provide methods for properly resetting the environment.

Player B's actions are defined to be 18 + the equivalent action value for Player A. For example, Player B's up action is `up` (20).

In addition to the regular ALE actions, the following (somewhat deprecated) actions are also processed by the FIFO interfaces:

save-state (43)	load-state (44)	system-reset (45)

7.2 Terminal States

Once the end of episode is reached (a terminal state in RL terminology), no further emulation takes place until the appropriate reset command is sent. This command is distinct from the Atari 2600 reset. This "system reset" avoids odd situations where the player can reset the game through button presses, or where the game normally resets itself after a number of frames. This makes for a cleaner environment interface. With the exception of the RL-Glue interface, which automatically resets the environment, the interfaces described here all provide a system reset command or method.

7.3 Saving and Loading

State saving and loading operates in a stack-based manner: each call to save stores the current environment state onto a stack, and each call to load restores the last saved copy and removes it from the stack. The ALE 0.2 save/load mechanism, provided for backward compatibility, instead overwrites its saved copy when a save is requested. When loading a state, the currently saved copy is preserved.

This functionality is provided in the fifo, shared library and CTypes interfaces. The shared library interface additionally provides state cloning/restoring capabilities.

7.4 Colour Averaging

Many Atari 2600 games display objects on alternating frames (sometimes even less frequently). This can be an issue for agents that do not consider the whole screen history. By default, *colour averaging* is not enabled. If enabled, the environment output (as observed by

agents) is a weighted blend of the last two frames. This behaviour can be turned on using the command–line argument `-color_averaging` (or the `setBool` function).

7.5 Action Repeat Stochasticity

Beginning with ALE 0.5.0, there is now an option (enabled by default) to add *action repeat stochasticity* to the environment. With probability p (default: $p = 0.25$), the previously executed action is executed again during the next frame, ignoring the agent's actual choice. This value can be modified using the option `action_repeat_probability`. The default value was chosen as the highest value for which human play-testers were unable to detect any delay or control lag.

The motivation for introducing action repeat stochasticity was to help separate *trajectory optimization* research from *robust controller optimization*, the latter often being the desired outcome in reinforcement learning (RL). We strongly encourage RL researchers to use the default stochasticity level in their agents, and clearly report the setting used. More details on the effects of action repeat stochasticity will be made available in future publications.

Note that, beginning with ALE 0.5.0, we have also removed the use of `rand()/srand()` from ALE. Agents' own randomization should therefore not affect the action repeat stochasticity.

7.6 Minimal Action Set

It may sometimes be convenient to restrict the agent to a smaller action set. This can be accomplished by querying the `RomSettings` class using the method `getMinimalActionSet`. This then returns a set of actions judged "minimal" to play a given game. Due to the potentially high impact of this setting on performance, we encourage researchers to clearly report the method used in their experiments.

8 Miscellaneous

This section provides additional relevant ALE information.

8.1 Audio and Visual (Screen) Output

ALE offers screen display and audio capabilities via the Simple DirectMedia Layer (SDL). Instructions on how to install the SDL library, as well as enabling SDL support within ALE, are provided in Section 2.2. Screen display can be enabled using the `display_screen` option (default: `false`), and sound playback using the `sound` option (default: `false`). SDL support has been tested under Linux and Mac OS X.

8.2 Recording Movies

ALE now provides support for recording frames; if sound is enabled (Section 8.1), it is also possible to record audio output. An example C++ program is provided which will record both visual and audio output for a single episode of play. This program is located at

`doc/examples/videoRecordingExample.cpp`

Compiling and running this program will create a directory `record`[3] in which frames will be saved sequentially and named according to their frame numbers. Thus, if the episode lasts 683 frames then the files `record/000000.png` to `record/000682.png` are created. Furthermore, sound output is also recorded as `record/sound.wav`. The following options control recording behaviour:

```
sound <true|false> -- whether to enable sound output (set to true for recording)
  default: false
record_screen_dir -- path to record screens; if empty, no recording occurs
  default: ""
record_sound_filename -- path to single wav file to be recorded;
          if empty, no recording occurs
  default: ""
```

Once frames and/or sound have been recorded, they may be joined into a movie file using the external program `ffmpeg` (installable on Mac OS X and most *nix systems through a package manager). For your convenience, two example scripts are provided:

- `doc/scripts/videoRecordingExampleJoinMacOSX.sh`
- `doc/scripts/videoRecordingExampleJoinUnix.sh`

These should be run from the same directory that the C++ example was run from. Unfortunately `ffmpeg` is a complicated beast and taming it may require tweaking specific to your system configuration. Please contact us if you would like to provide an example script for a different configuration.

Here is a full, step-by-step example on Mac OS X (after building the project):

```
> cd ale_0_5
ale_0_5> doc/examples/videoRecordingExample space_invaders.bin

A.L.E: Arcade Learning Environment (version 0.5.0)
[Powered by Stella]

[ usual ALE output ]
```

[3]The example program creates this directory, using a system call to `mkdir`. If this fails on your machine, you will need to manually create this directory.

```
Recording screens to directory: record

Recording complete. See manual for instructions on creating a video.

ale_0_5> doc/scripts/videoRecordingExampleJoinMacOSX.sh

ffmpeg version 2.6.1 Copyright (c) 2000-2015 the FFmpeg developers
  built with Apple clang version 4.1 (tags/Apple/clang-421.11.65) ...

[ loads of output ]

ale_0_5> open agent.mov
```

9 Troubleshooting / FAQ

More questions and answers may be found on the ALE-users mailing list:

`https://groups.google.com/forum/#!forum/arcade-learning-environment`

- I downloaded ALE and I installed it successfully but I cannot find any ROM file at `roms/`. Do I have to get them somewhere else?

  Yes. We do not distribute Atari 2600 ROMs.

- The C++ examples compile, but when run give the following error:

  `dyld: Library not loaded: libale.so .`

  You may need to add ALE to your library path:

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/ale_0_5
  ```

  or, under Mac OS X,

  ```
  export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/path/to/ale_0_5
  ```

  Alternatively, run the examples from the directory in which `libale.so` resides.

- I am trying to run ALE via command line (FIFO interface, or RL-Glue agent) but I keep getting an "unsupported ROM file" error. What am I doing wrong?

  These errors are generally hard to be solved in every possible way, but our experience has shown several people forgetting to specify the ROM file as the last argument on the command line, in lower case. Moreover, each game is tied to an internal parser which relies on a specific filename (e.g. `pong.bin` for Pong). This is specified in the header file (hpp) for the corresponding parser (e.g. `src/games/supported/Pong.hpp`). You may want to check whether the ROM you are trying to load is supported by the current version of ALE.

- I am having problems when compiling with the option `USE_SDL=ON` on Mac OS X. The compiler complains it was not able to find the proper SDL files. What should I do?

  We spent a considerable amount of time trying to make sure CMake would properly find the correct path in different situations. Something we realized is that sometimes, if the user has two installations (generally one via `brew` and another via `dmg` package), conflicts may arise. Generally it is enough to set the variable `SDL_LIBRARY` as below (the most common path where SDL is installed is used here):

  ```
  SDL_LIBRARY:STRING=/usr/local/lib/libSDLmain.a;/usr/local/lib/libSDL.dylib;
  -framework Cocoa
  ```

- I want to be able to extract from the game the number of lives my agent still has. How can I do it?

  Previous versions of ALE did not support this. We started to support such feature since version 0.5.0, through the use of the function `lives()`.

- When extracting the screen I realized that I never see a pixel with an odd value. However, the pixel is represented as a byte. Shouldn't it be up to 255 with odd an even values?

  No, the Atari 2600 console (NTSC format) only supports 128 colours. Therefore, even though the colours are represented in a byte, in fact only the seven most significant bits are used. Consequently you have to right-shift the colour byte by 1 bit to produce consecutively numbered colour values.

- When trying to run ALE I get a linking error stating `-lrlagent` could not be found. What should I do?

  The `rlragent` is generated by the library RL-Glue. Therefore, we recommend you to verify your installation of RL-Glue and to make sure it is being properly found when compiling ALE code. If RL-Glue support is not needed, consider running cmake with `USE_RLGLUE=OFF` (or disabling it in the makefile).

- When running/compiling my own C++ agent with `display_screen`/SDL enabled, I get the following error(s):

  ```
  Terminating app due to uncaught exception 'NSInternalInconsistencyException',
  reason: 'Error (1000) creating CGSWindow on line 259'

  (and/or)

  Undefined symbols for architecture x86_64:
    "_SDL_main", referenced from:
        -[SDLMain applicationDidFinishLaunching:] in libSDLmain.a(SDLMain.o)
  ld: symbol(s) not found for architecture x86_64
  ```

  Make sure your code contains `#include <SDL.h>` and that `__USE_SDL` is defined. Also, compiling under Mac OS X requires the inclusion of the flag `-framework Cocoa`. See the shared library example for a working example.

- The Python interface isn't working! It can't find `libale_c.so`.

  Make sure to compile ALE using CMake. This will build the library required by the Python interface.

10 ALE Interface Specifications

Below are listed the functions available in ALE interface with the description of their behaviour. The functions were divided in different sections to make the presentation more clear.

10.1 Initialization

`ALEInterface(bool display_screen)`: ALE constructor. If the `display_screen` parameter is set to `true`, and ALE was compiled with SDL, the game display will be presented. If set to `false` one will not see the game being played.

`void loadROM(string rom_file)`: Resets ALE and then loads a game. After this call the game should be ready to play. If one changes (or sets) a setting (Section 10.2), it is necessary to call this function after the change so it can take effect.

10.2 Parameters setting and retrieval

`string getString(const string& key)`: Gets the value of any flag passed as parameter that has a string value; *e.g.*: `getString(''record_sound_filename'')`.

`int getInt(const string& key)`: Gets the value of any flag passed as parameter that has an integer value; *e.g.*: `getInt(''frame_skip'')`.

`bool getBool(const string& key)`: Gets the value of any flag passed as parameter that has a boolean value; *e.g.*: `getBool(''restricted_action_set'')`.

`float getFloat(const string& key)`: Gets the value of any flag passed as parameter that has a float value; *e.g.*: `getBool(''repeat_action_probability'')`.

`void setString(const string& key, const string& value)`: Sets the value of any flag that has a string type; *e.g.*: `setString("record_screen_dir", "record")`. `loadRom()` must be called before the setting will take effect.

`void setInt(const std::string& key, const int value)`: Sets the value of any flag that has an integer type; *e.g.*: `setInt("frame_skip", 1)`. `loadRom()` must be called before the setting will take effect.

`void setBool(const std::string& key, const bool value)`: Sets the value of any flag that has a boolean type; *e.g.*: `setBool("restricted_action_set", false)`. `loadRom()` must be called before the setting will take effect.

`void setFloat(const std::string& key, const float value)`: Sets the value of any flag that has a float type; *e.g.*: `setFloat("repeat_action_probability", 0.25)`. `loadRom()` must be called before the setting will take effect.

10.3 Acting and Perceiving

`reward_t act(Action action)`: Applies an action to the game and returns the reward. It is the user's responsibility to check if the game has ended and reset when necessary (this method will

keep pressing buttons on the game over screen).

`bool game_over()`: Indicates if the game has ended.

`void reset_game()`: Resets the game, but not the full system (it is not "equivalent" to unplug the console from electricity).

`ModeVect getAvailableModes()`: Returns the vector of modes available for the current game. This should be called only after the ROM is loaded.

`void setMode(game_mode_t m)`: Sets the mode of the game. The mode must be an available mode (otherwise it throws an exception). This should be called only after the ROM is loaded.

`DifficultyVect getAvailableDifficulties()`: Returns the vector of difficulties available for the current game. This should be called only after the ROM is loaded.

`void setDifficulty(difficulty_t m)`: Sets the difficulty of the game. The difficulty must be an available mode (otherwise it throws an exception). This should be called only after the ROM is loaded.

`ActionVect getLegalActionSet()`: Returns the vector of legal actions (all the 18 actions). This should be called only after the ROM is loaded.

`ActionVect getMinimalActionSet()`: Returns the vector of the minimal set of actions needed to play the game (all actions that have some effect on the game). This should be called only after the ROM is loaded.

`int getFrameNumber()`: Returns the current frame number since the loading of the ROM.

`const int lives()`: Returns the agent's remaining number of lives. If the game does not have the concept of lives (*e.g.* FREEWAY), this function returns 0.

`int getEpisodeFrameNumber()`: Returns the current frame number since the start of the current episode.

`const ALEScreen &getScreen()`: Returns a matrix containing the current game screen.

`void getScreenGrayscale(pixel_t *grayscale_output_buffer)`: This method should receive an array of length width $\times$ height (generally $160 \times 210 = 33,600$) and then it will fill this array with the grayscale colours

`void getScreenRGB(pixel_t *output_rgb_buffer)`: This method should receive an array of length 3$\times$ width $\times$ height (generally $3 \times 160 \times 210 = 100,800$) and then it will fill this array with the RGB colours. The first positions contain the red colours, followed by the green colours and then the blue colours

`const ALERAM &getRAM()`: Returns a vector containing current RAM content (byte-level).

`void saveState()`: Saves the current state of the system if one wants to be able to recover a state in the future; *e.g.* in search algorithms.

`void loadState()`: Loads a previous saved state of the system once we have a state saved.

`ALEState cloneState()`: Makes a copy of the environment state. This copy does **not** include pseudo-randomness, making it suitable for planning purposes.

`ALEState cloneSystemState()`: This makes a copy of the system and environment state, suitable for serialization. This includes pseudo-randomness and so is **not** suitable for planning purposes.

`void restoreState(const ALEState& state)`: Reverse operation of `cloneState()`. This does not restore pseudo-randomness, so that repeated calls to `restoreState()` in the stochastic controls setting will not lead to the same outcomes. By contrast, see `restoreSystemState`.

`void restoreSystemState(const ALEState& state)`: Reverse operation of `cloneSystemState`.

10.4 Recording trajectories

`void saveScreenPNG(const string& filename)`: Saves the current screen as a `png` file.

`ScreenExporter *createScreenExporter(const string &path) const`: Creates a ScreenExporter object which can be used to save a sequence of frames. Frames are saved in the directory 'path', which needs to exists. This is used to generate movies depicting the behavior of agents.

11 Command-line Arguments

Command-line arguments are passed to ALE before the ROM filename. These are converted into options (minus the prefix hyphen ('-') within ALE. Currently, setting options from the command-line is only relevant to the fifo and RL-Glue interfaces. When using the C++ or Python interface, one should instead directly invoke the relevant setter functions (e.g. `setInt()`; see Section 10.2). The configuration file `ale_0_5/stellarc` can also be used to set frequently used options.

11.1 Main Arguments

```
-help -- prints out help information

-game_controller <fifo|fifo_named|rlglue> -- selects an ALE interface
  default: unset

-random_seed <###> -- picks the ALE random seed; if set to 0, sets to current
  time instead
  default: 0

-display_screen <true|false> -- if true and SDL is enabled, displays ALE screen
  default: false

-sound  <true|false> -- if true and SDL is enabled, the game will have game
  sounds enabled
  default: false
```

11.2 Environment Arguments

```
-max_num_frames ### -- max. total number of frames, or 0 for no maximum
  (it is not available in the shared library interface, i.e. to be set
  by C++  or Python code directly linking to the shared library)
  default: 0

-max_num_frames_per_episode ### -- max. number of frames per episode
```

```
  default: 0

-frame_skip ### -- frame skipping rate; 1 indicates no frame skip
  default: 1

-color_averaging <true|false> -- if true, enables colour averaging
  default: false

-record_screen_dir [save_directory] -- saves game screen images to
  save_directory

-repeat_action_probability -- stochasticity in the environment. It is the
  probability the previous action will repeated without executing the new
  one
  default: 0.25
```

11.3 FIFO Interface Arguments

```
-run_length_encoding <true|false> -- if true, encodes data using run-length
  encoding
  default: true
```

11.4 RL-Glue Interface Arguments

```
-send_rgb <true|false> -- if true, RGB values are sent for each pixel
  instead of the pallette index values
  default: false

-restricted_action_set <true|false> -- if true, agents use a smaller set of
  actions (RL-Glue interfaces only)
  default: false
```