

# Arcade Learning Environment Technical Manual (v.0.5.0)

June 14, 2015

# Contents

<b>1</b>	<b>Overview</b>	<b>4</b>
<b>2</b>	<b>Installing</b>	<b>4</b>
2.1	Requirements . . . . .	4
2.2	Installation/Compilation . . . . .	4
<b>3</b>	<b>C++ Agent: The Shared Library Interface</b>	<b>5</b>
<b>4</b>	<b>Implementing an agent in Python (shared libraries)</b>	<b>7</b>
<b>5</b>	<b>FIFO Interface</b>	<b>8</b>
5.1	Handshaking . . . . .	8
5.2	Main Loop – ALE . . . . .	8
5.2.1	RAM_string . . . . .	9
5.2.2	screen_string . . . . .	9
5.2.3	episode_string . . . . .	9
5.2.4	Example . . . . .	9
5.3	Main Loop – Agent . . . . .	9
5.4	Termination . . . . .	10
<b>6</b>	<b>RL-Glue Interface</b>	<b>10</b>
6.1	Sample Agent and Experiment . . . . .	10
6.2	Actions and Observations . . . . .	11
<b>7</b>	<b>Environment Specifications</b>	<b>11</b>
7.1	Available Actions . . . . .	11
7.2	Terminal States . . . . .	12
7.3	Saving and Loading States . . . . .	12
7.4	Colour Averaging . . . . .	12
7.5	Randomness . . . . .	12
7.6	Minimal Action Set . . . . .	13
<b>8</b>	<b>ALE Interface Specifications</b>	<b>13</b>
8.1	Initialization . . . . .	13
8.2	Parameters setting and retrieval . . . . .	13
8.3	Acting and Perceiving . . . . .	14
8.4	Recording trajectories . . . . .	15
<b>9</b>	<b>Command-line Arguments</b>	<b>15</b>
9.1	Main Arguments . . . . .	15
9.2	Environment Arguments . . . . .	15
9.3	FIFO Interface Arguments . . . . .	16

9.4	RL-Glue Interface Arguments . . . . .	16
<b>10</b>	<b>Miscellaneous</b>	<b>16</b>
10.1	Displaying the Screen . . . . .	16
10.2	Recording Movies . . . . .	17
<b>11</b>	<b>Troubleshooting / FAQ</b>	<b>18</b>

# 1 Overview

This document is roughly divided into three parts. The first part describes how to install the Arcade Learning Environment. 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.

## 2 Installing

### 2.1 Requirements

The basic requirements to build and run the ALE are:

- CMake or make
- A C++ compiler

Beginning with version 0.5, 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 the ALE requirements before compiling the 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<sup>1</sup>):

```
> 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 the 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 later 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 11) if compilation fails, e.g. because of a missing `SDL.h`.

### 3 C++ Agent: The 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:

```
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 9. Just as an example, to set the environment's seed we write `ale.setInt("random_seed", 123);`

---

<sup>1</sup><http://brew.sh>

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 <iostream>  
#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;  
    while (!ale.game_over()) {  
        Action a = legal_actions[rand() % legal_actions.size()];
```

```

        float reward = ale.act(a);
        totalReward += reward;
    }
    cout << "The episode ended with score: " << totalReward << endl;
}
return 0;
}

```

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

`sharedLibraryExample.cpp`

example for more details, including compilation, as well as Section 11 if errors arise. A complete list of functions available in the class `ALEInterface` is given in Section 8.

## 4 Implementing an agent in Python (shared libraries)

To use the Python interface it is necessary to install it after the ALE was compiled. If one has root access to the machine it is enough to run:

```
pip install .
```

otherwise, if he does not have root access, the following should work:

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

Then, to use it in a Python code it is enough to import it properly, for example, as in the example below (`from ale_python_interface import ALEInterface`). A code containing an example is available at `doc/examples/python_example.py`.

Aside from a few minor differences, the Python interface mirrors the C++ interface. A complete example in Python is below:

```

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 × 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  $2 \times \text{www} \times \text{hhh}$  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<sup>2</sup>.

In order to use the RL-Glue interface, ALE must be compiled with RL-Glue support. This is achieved by setting `option(USE_RLGLUE "Use RL-Glue" ON)` in the `CMakeListst.txt` file.

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

### 6.1 Sample Agent and Experiment

Example source code can be found under

`ale_0_4/doc/examples`

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

```
make rlg glueAgent
```

As with any RL-Glue application, you will need to start the following processes to run the sample RL-Glue agent in ALE:

---

<sup>2</sup><http://glue.rl-community.org>

- `rl_glue`
- `RLGlueAgent`
- `RLGlueExperiment`
- `ale` (with command-line argument `-game_controller rlglue`)

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` command-line argument was set to `true`.

The observation space depends on whether the `-send_rgb` argument was passed to ALE. If it was not passed, or it was set to `false`, the observation space consists of 33,728 integers: first the 128 bytes of RAM (taking values in 0–255), then the 33,600 screen pixels (taking value in 0–127). The screen is provided in row-order, i.e. beginning with the 160 pixels that compose the first row.

If `-send_rgb` was set to `true` on the command-line, the observation space consists of 100,928 integers: first the same 128 bytes of RAM, followed by 100,800 bytes describing the screen. Each pixel is described by three bytes, taking values from 0–255, specifying the pixel’s red, green and blue components in that order. The pixel order is the same as in the default case.

## 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:

<code>noop</code> (0)	<code>fire</code> (1)	<code>up</code> (2)	<code>right</code> (3)	<code>left</code> (4)
<code>down</code> (5)	<code>up-right</code> (6)	<code>up-left</code> (7)	<code>down-right</code> (8)	<code>down-left</code> (9)
<code>up-fire</code> (10)	<code>right-fire</code> (11)	<code>left-fire</code> (12)	<code>down-fire</code> (13)	<code>up-right-fire</code> (14)
<code>up-left-fire</code> (15)	<code>down-right-fire</code> (16)	<code>down-left-fire</code> (17)	<code>reset*</code> (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. In general it should be replaced by either a call to `StellaEnvironment::reset` or by sending the `system_reset` command, depending on the interface.

Player B’s actions are the same as those of Player A, but with 18 added. For example, Player B’s up action corresponds to the integer 20.

In addition to the regular ALE actions, the following 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 States

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.

## 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 going to be 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 Randomness

The Atari 2600 games, as emulated by Stella, are deterministic. However, previous works have shown that it is of interest to add stochasticity to this domain, to avoid the exploration of open loop planning (trajectory optimization), *i.e.* to prevent agents from simply learning a trajectory by rote.

Such stochasticity is implemented in the ALE through the performed actions. With probability  $p$  (default:  $p = 0.25$ ) the previous executed action is executed during the next frame, ignoring the agent’s actual choice of action. The motivation of this implementation is to resemble how humans play games. Humans are not capable of being extremely precise

regarding which frame they will take an action. Because of that we thought it would be a reasonable approach to implement stochasticity in such a way.

i Also, it is important to stress that we have added TinyMT as the random number generator. This was made to avoid any sort of problem one could have by overriding the environment's seed with later calls (this can happen with `rand()/srand()`). In practice this has no effect to those using the ALE, but it is important to stress such feature.

## 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 on the method used in their experiments.

# 8 ALE Interface Specifications

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

## 8.1 Initialization

`ALEInterface(bool display_screen)`: ALE constructor. If the `display_screen` parameter is set to `true`, and the 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 the ALE and then loads a game. After this call the game should be ready to play. If one changes (or sets) a setting (Section 8.2), it is necessary to call this function after the change so it can take effect.

## 8.2 Parameters setting and retrieval

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

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

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

`float getFloat(const string& key)`: Get 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):` Set the value of any flag that has a string type; *e.g.*: `setString("random_seed", "time"). loadRom()` must be called before the setting will take effect.

`void setInt(const std::string& key, const int value):` Set 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):` Set 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):` Set 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.

### 8.3 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).

`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.

`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.

## 8.4 Recording trajectories

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

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

## 9 Command-line Arguments

Command-line arguments are passed to ALE before the ROM filename. The current version removed all the internal agents in the game, therefore, the command-line arguments are mostly useful to start the communication when using FIFO pipes.

However, these parameters can also be set when using the C++ or Python interface, for example. In this case, they are set using those functions discussed in Section 8.2. The configuration file `ale_0_5/stellarc` can also be used to set frequently used command-line arguments.

The parameters that can be set are all listed below.

### 9.1 Main Arguments

```
-help -- prints out help information

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

-random_seed <###|time> -- picks the ALE random seed, or sets it to current time
    default: time

-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
```

### 9.2 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 be repeated without executing the new one  
default: 0.25

### 9.3 FIFO Interface Arguments

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

### 9.4 RL-Glue Interface Arguments

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

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

## 10 Miscellaneous

This section provides additional relevant ALE information.

### 10.1 Displaying the Screen

ALE offers screen display and audio capabilities via the Simple DirectMedia Layer (SDL). Instructions on how to install the SDL library are provided in Section 2.2.



To compile with SDL support, you should change the file `CMakeLists.txt`, replacing the word `OFF` by `ON` in the line `option(USE_SDL "Use SDL" OFF)`. Then, screen display can be enabled with the `display_screen` argument (command line or function `setBool`).

SDL support has been tested under Linux and Mac OS X.

## 10.2 Recording Movies

ALE now contains support for recording frames and sound via SDL (Section 10.1). An example C++ program is provided which will record a single episode of play. This program is located at

`doc/examples/videoRecordingExample.cpp`

Compiling and running this program will create a directory `record`<sup>3</sup> 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 flags control recording behaviour:

```
-display_screen <true|false> -- should be set to true for recording
    default: false
-sound <true|false> -- whether to enable sound output
    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 OSes 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_4
ale_0_4> doc/examples/videoRecordingExample space_invaders.bin
```

---

<sup>3</sup>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.

A.L.E: Arcade Learning Environment (version 0.4.4)  
[Powered by Stella]

[ usual ALE output ]

Recording screens to directory: record

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

ale\_0\_4> 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\_4> open agent.mov

## 11 Troubleshooting / FAQ

- I downloaded the 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 have distribution rights over any Atari 2600 ROM. Therefore, they are not in our repository nor in the installation package. There are websites that distribute such files, supposedly with the proper copyrights.

- The C++ examples compile, but when run give the following error: `dyld: Library not loaded: libale.so`. Help!

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 the 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 supposed to have a proper name for the binary file. This is specified in the header file (`hpp`) of the game. Finally, it is also important to check whether the ROM you are trying to load is supported by the current version of the ALE.

- I am having problems when compiling with the option `USE_SDL ON` in the Mac OS. 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 the 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 and even values?

No, the Atari 2600 console only had 128 colours. Therefore, even though the colours are represented in a byte, in fact only 7 bits are used. The least significative bit is always zero, consequently you have to shift the byte `>> 1` to obtain the correct colour value.

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

The `rlagent` 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 the ALE code.