Arcade Learning Environment
Technical Manual

May 3, 2015

Contents

1 Overview

This document describes how to install the Arcade Learning Environment as well as its technical details, including command-line flags, how to use the shared library interface, and the FIFO protocol for agents communicating with ALE via stdin/stdout.

The following ways of interfacing with ALE are detailed in this document:

- **Shared library interface.** Loads ALE as a shared library (Section 4).
- **FIFO interface.** Communicates with ALE through a text interface (Section 5).
- **RL-Glue interface.** Communicates with ALE via RL-Glue (Section 6).

2 Installing

2.1 Requirements

To build and run ALE, you will need:

- g++ / make

2.2 Installation/Compilation

This tutorial assumes that you have extracted ALE to `ale_0_4`. Compiling ALE on a UNIX machine is as simple as:

```
> cd ale_0_4
ale_0_4> cp makefile.unix makefile
ale_0_4> make
```

Alternate makefiles are provided for

- Mac OS X: `makefile.mac`

2.3 Sample Agents

The following sample agents are available:

- **Java agents / FIFO interface.** Refer to the accompanying ALE Java Agent tutorial.
- **C++ agent / shared library interface.** See Section 4 and `ale_0_4/doc/examples/sharedLibraryInterfaceExample.cpp`.
- **C++ agent / RL-Glue interface.** See Section 6 and `ale_0_4/doc/examples/RLGlueAgent.c`.

3 Command-line Arguments

Command-line arguments are passed to ALE before the ROM filename. TODO: Provide a different example to use the command-line arguments (if any, since we removed the internal agent).

The configuration file `ale_0_4/stellarc` can also be used to set frequently used command-line arguments.

3.1 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
```

3.2 Environment Arguments

```
-max_num_frames ### -- max. total number of frames, or 0 for no maximum
    (not in shared library interface)
  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

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

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

3.3 FIFO Interface Arguments

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

3.4 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
```

4 Shared Library Interface

The shared library interface allows agents to directly access ALE via a class called `ALEInterface`, defined in `ale_interface.hpp`.

4.1 Sample Agent

Example source code can be found under

```
ale_0_4/doc/examples/sharedLibraryInterfaceExample.cpp
```

Assuming you installed ALE under `/home/marc/ale_0_4`, the shared library agent can be compiled with the following command:

```
g++ -I/home/marc/ale_0_4/src -L/home/marc/ale_0_4
  -lale -lz sharedLibraryInterfaceExample.cpp
```

or alternatively by appropriately setting paths in `ale_0_4/doc/examples/Makefile.sharedlibrary` before running

```
ale_0_4/doc/examples> make sharedLibraryAgent
```

To run the agent, you may need to add ALE to your library path:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/marc/ale_0_4
```

or under Mac OS X,

```
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/home/marc/ale_0_4
```

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 × `www` × `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[1].

In order to use the RL-Glue interface, ALE must be compiled with RL-Glue support. This is achieved by setting `USE_RL=1` in the makefile.

Specifying the command-line argument `-game_controller rlglue` 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 `/home/marc/ale_0_4`, the RL-Glue agent and experiment can be compiled with the following command:

```
make rlglueAgent
```

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

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

Please refer to the RL-Glue documentation for more details.

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

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:

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. 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 enabled: the environment output (as observed by agents) is a weighted blend of the last two frames. This behaviour can be turned off using the command-line argument `-disable_colour_averaging`.

7.5 Randomness

The Atari 2600 games, as emulated by Stella, are deterministic. Sometimes it may be of interest to add a tiny amount of stochasticity to prevent agents from simply learning a trajectory by rote. This is done **TODO**.

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. Of course, algorithm designers are encouraged to devise algorithms that don't depend on this minimal action set for success.

8 Miscellaneous

This section provides additional relevant ALE information.

8.1 Displaying the Screen

ALE offers screen display capabilities via the Simple DirectMedia Layer (SDL). This requires the following libraries:

- **libsdl**
- **libsdl-gfx**
- **libsdl-image**

To compile with SDL support, you should set `USE_SDL=1` in the ALE makefile. Then, screen display can be enabled with the `-display_screen` command-line argument.

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

8.2 Recording Movies

ALE now contains support for recording frames and sound via SDL (Section 8.1). 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`[2] 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

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

[ usual ALE output ]

Recording screens to directory: record
```

[2]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 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
```