Half Field Offense
Technical Manual

Matthew Hausknecht

April 3, 2015

Contents

1 Overview

This document describes the state and action spaces of the HFO domain.

2 State Space

The state features used by HFO are designed with the mindset of providing an overcomplete, basic, egocentric viewpoint. The features are basic in the sense that they provide distances and angles to relevant points of interest, but do not include higher level perceptions such as the largest angle between a goal post and goalkeeper.

All features are encoded as floating point values normalized to the range of [-1,1]. Different types of features are discussed next.

2.1 Boolean Features

Boolean features assume either the minimum feature value of -1 or the maximum feature value of 1.

2.2 Valid Features

Since feature information is attained from the Agent's world-model, it is possible that, the world model's information may be stale or incorrect. *Valid features* are boolean features indicating consistency of world model predictions. For example, if the world model's estimate of the agent's position is known to be flawed, the *valid feature* for self position would assume the minimum value of -1. Otherwise it will assume the maximum value of 1.

The features associated with a valid feature are given the value of zero if an inconsistency is detected. For example, if the world model detects that the agent's velocity is invalid, the feature that encodes the magnitude of self velocity will be set to zero.

2.3 Angular Features

Angular features (e.g. the angle to the ball), are encoded as two floating point numbers – the $sin(\theta)$ and $cos(\theta)$ where θ is the original angle.

This encoding allows the angle to vary smoothly for all possible angular values. Other encodings such as radians or degrees have a discontinuity that when normalized, could cause the feature value to flip between the maximum and minimum value in response to small changes in θ.

2.4 Distance Features

Distance features encode the distance to objects of interest. Unless otherwise indicated, they are normalized against the maximum possible distance in the HFO playfield (defined as $\sqrt{l^2 + w^2}$ where l, w are the length and width of the HFO playfield). A distance of zero will be encoded with the minimum feature value of -1 while a maximum distance will be encoded with 1.

2.5 Landmark Features

Landmark features encode the relative angle and distance to a landmark of interest. Each landmark feature consists of three floating point values, two to encode the angle to the landmark and one to encode the distance. Note that if the agent's self position is invalid, then the landmark feature values are zeroed.

2.6 Player Features

Player features are used to encode the relationship of the agent to another player or opponent. Each player feature is encoded as 1) a landmark feature of that player's location 2) the global angle of that player's body 3) the magnitude of the player's velocity and 4) the global angle of the player's velocity. Eight floating point numbers are used to encode each player feature.

2.7 Other Features

Some features, such as the agent's stamina, do not fall into any of the above categories. These features are referred to as *other features*.

3 State Feature List

Basic Features are always present and independent of the number of teammates or opponents. The 32 basic features are encoded using 58 floating point values (*angular features* require two floats, *landmark features* require 3). Additionally a variable number of *player features* are then added. This number depends on the number of teammates and opponents in the HFO game, but 8 floating point values are required for each player feature. Thus, the total number of features is $58 + 8 * \text{num\_teammates} + 8 * \text{num\_opponents}$.

- **Self_Pos_Valid** [Valid] Indicates if self position is valid.
- **Self_Vel_Valid** [Valid] Indicates if the agent's velocity is valid.
- **Self_Vel_Ang** [Angle] Angle of agent's velocity.
- **Self_Vel_Mag** [Other] Magnitude of agent's velocity. Normalized against the maximum observed self speed, 0.46.
- **Self_Ang** [Angle] Agent's Global Body Angle.
- **Stamina** [Other] Agent's Stamina: The amount of remaining stamina the agent has. Normalized against the maximum observed agent stamina of 8000.
- **Frozen** [Boolean] Indicates if the agent is Frozen. Frozen status can happen when being tackled by another player.
- **Colliding_with_ball** [Boolean] Indicates if the agent is colliding with the ball.
- **Colliding_with_player** [Boolean] Indicates if the agent is colliding with another player.
- **Colliding_with_post** [Boolean] Indicates if the agent is colliding with a goal post.
- **Kickable** [Boolean] Indicates if the agent is able to kick the ball.
- **Goal Center** [Landmark] Center point between the goal posts.
- **Goal Post Top** [Landmark] Top goal post.
- **Goal Post Bot** [Landmark] Bottom goal post.
- **Penalty Box Center** [Landmark] Center of the penalty box line.
- **Penalty Box Top** [Landmark] Top corner of the penalty box.
- **Penalty Box Bot** [Landmark] Bottom corner of the penalty box.
- **Center Field** [Landmark] The left middle point of the HFO play area. True center of the full-field.

- **Corner Top Left** [Landmark] Top left corner HFO Playfield.
- **Corner Top Right** [Landmark] Top right corner HFO Playfield.
- **Corner Bot Right** [Landmark] Bot right corner HFO Playfield.
- **Corner Bot Left** [Landmark] Bot left corner HFO Playfield.
- **OOB Left Dist** [Distance] Distance to the nearest point of the left side of the HFO playable area. E.g. distance remaining before the agent goes out of bounds in left field.
- **OOB Right Dist** [Distance] Distance remaining before the agent goes out of bounds in right field.
- **OOB Top Dist** [Distance] Distance remaining before the agent goes out of bounds in top field.
- **OOB Bot Dist** [Distance] Distance remaining before the agent goes out of bounds in bottom field.
- **Ball Pos Valid** [Valid] Indicates if the ball position estimate is valid.
- **Ball Angle** [Angle] Angle to the ball from the agent's perspective.
- **Ball Dist** [Distance] Distance to the ball.
- **Ball Vel Valid** [Valid] Indicates if the ball velocity estimate is valid.
- **Ball Vel Mag** [Other] Global magnitude of the ball velocity. Normalized against the observed maximum ball velocity, 3.0.
- **Ball Vel Ang** [Angle] Global angle of ball velocity.
- **Teammate Features** [Player] One teammate feature for each teammate active in HFO, sorted by proximity to the agent.
- **Opponent Features** [Player] One opponent feature for each opponent present, sorted by proximity to the player.

4 Action Space

The action space of the HFO domain is primitive: basic parameterized actions are provided for locomotion and kicking. Control of the agent's head and gaze is not provided. The primitive actions are as follows:

- **Dash**(power, degrees): Moves the agent with power [-100, 100] where negative values move backwards. The relative direction of movement is given in degrees and varies between [-180,180] with 0 degrees being a forward dash and 90 degrees dashing to the agent's right side. Note, dashing does not turn the agent.
- **Turn**(degrees): Turns the agent in the specified direction. Valid values range between [-180, 180] degrees where 90 degrees turns the agent to directly to its right side.
- **Tackle**(degrees): Contest the ball. Direction varies between [-180, 180]. TODO: Better description.
- **Kick**(power, degrees): Kick the ball with power [0, 100] in relative direction [-180, 180]. Has no effect if the agent does not possess the ball.
- **Quit**: Indicates to the agent server that you wish to terminate the HFO environment.

5 Installing

5.1 Requirements

To build and run ALE, you will need:

- g++ / make

5.2 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`

5.3 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 **??** and `ale_0_4/doc/examples/sharedLibraryInterfaceExample.cpp`.
- **C++ agent / RL-Glue interface.** See Section **??** and `ale_0_4/doc/examples/RLGlueAgent.c`.

6 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.

6.1 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
```

6.2 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
```

6.3 FIFO Interface Arguments

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

6.4 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
```

7 Shared Library Interface

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

7.1 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
```

8 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.

8.1 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.

8.2 Main Loop – ALE

After handshaking, ALE will then loop until one of the termination conditions occurs; these conditions are described below in Section **??**. 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.

8.2.1 `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 ':'.

8.2.2 `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.

8.2.3 `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.

8.2.4 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
```

8.3 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 **??** for the list of available actions.

8.4 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.

9 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.

9.1 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

9.2 Actions and Observations

The action space consists of both Player A and Player B's actions (see Section ?? 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.

10 Environment Specifications

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

10.1 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)

10.2 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.

10.3 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.

10.4 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`.

10.5 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**.

10.6 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.

11 Miscellaneous

This section provides additional relevant ALE information.

11.1 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.