Gym Retro Documentation

OpenAI

Aug 30, 2020

Contents:

Status: Maintenance (expect bug fixes and minor updates)

Gym Retro lets you turn classic video games into Gym environments for reinforcement learning and comes with integrations for ~1000 games. It uses various emulators that support the Libretro API, making it fairly easy to add new emulators.

Supported platforms:

- Windows 7, 8, 10
- macOS 10.13 (High Sierra), 10.14 (Mojave)
- Linux (manylinux1)

Supported Pythons:

- 3.6
- 3.7
- 3.8

Each game integration has files listing memory locations for in-game variables, reward functions based on those variables, episode end conditions, savestates at the beginning of levels and a file containing hashes of ROMs that work with these files.

Please note that ROMs are not included and you must obtain them yourself. Most ROM hashes are sourced from their respective No-Intro SHA-1 sums.

CHAPTER 1

Documentation

Documentation is available at https://retro.readthedocs.io/en/latest/

You should probably start with the Getting Started Guide.

CHAPTER 2

Contributing

See CONTRIBUTING.md

CHAPTER 3

Changelog

See CHANGES.md

CHAPTER 4

Emulated Systems

- Atari
  - Atari2600 (via Stella)
- NEC
  - TurboGrafx-16/PC Engine (via Mednafen/Beetle PCE Fast)
- Nintendo
  - Game Boy/Game Boy Color (via gambatte)
  - Game Boy Advance (via mGBA)
  - Nintendo Entertainment System (via FCEUmm)
  - Super Nintendo Entertainment System (via Snes9x)
- Sega
  - GameGear (via Genesis Plus GX)
  - Genesis/Mega Drive (via Genesis Plus GX)
  - Master System (via Genesis Plus GX)

See LICENSES.md for information on the licenses of the individual cores.

CHAPTER 5

Included ROMs

The following non-commercial ROMs are included with Gym Retro for testing purposes:

- the 128 sine-dot by Anthrox
- Sega Tween by Ben Ryves
- Happy 10! by Blind IO
- 512-Colour Test Demo by Chris Covell
- Dekadrive by Dekadence
- Automaton by Derek Ledbetter
- Fire by dox
- FamiCON intro by dr88
- Airstriker by Electrokinesis
- Lost Marbles by Vantage

CHAPTER 6

Citation

Please cite using the following BibTeX entry:

```
@article{nichol2018retro,
  title={Gotta Learn Fast: A New Benchmark for Generalization in RL},
  author={Nichol, Alex and Pfau, Vicki and Hesse, Christopher and Klimov, Oleg and␣
↪Schulman, John},
  journal={arXiv preprint arXiv:1804.03720},
  year={2018}
}
```

6.1 Getting Started

Gym Retro requires one of the supported versions of Python (3.5, 3.6, or 3.7). Please make sure to install the appropriate distribution for your OS beforehand. Please note that due to compatibility issues with some of the cores, 32-bit operating systems are not supported.

```
pip3 install gym-retro
```

See the section *Development* if you want to build Gym Retro yourself (this is only useful if you want to change the C++ code, not required to integrate new games).

6.1.1 Create a Gym Environment

After installing you can now create a Gym environment in Python:

```
import retro
env = retro.make(game='Airstriker-Genesis')
```

`Airstriker-Genesis` has a non-commercial ROM that is included by default.

Please note that other ROMs are not included and you must obtain them yourself. Most ROM hashes are sourced from their respective No-Intro SHA-1 sums. See *Importing ROMs* for information about importing ROMs into Gym Retro.

6.1.2 Example Usage

Gym Retro is useful primarily as a means to train RL on classic video games, though it can also be used to control those video games from Python.

Here are some example ways to use Gym Retro:

Interactive Script

There is a Python script that lets you interact with the game using the Gym interface. Run it like this:

```
python3 -m retro.examples.interactive --game Airstriker-Genesis
```

You can use the arrow keys and the `X` key to control your ship and fire. This Python script lets you try out an environment using only the Gym Retro Python API and is quite basic. For a more advanced tool, check out the *The Integration UI*.

Random Agent

A random agent that chooses a random action on each timestep looks much like the example random agent for Gym:

```
import retro

def main():
    env = retro.make(game='Airstriker-Genesis')
    obs = env.reset()
    while True:
        obs, rew, done, info = env.step(env.action_space.sample())
        env.render()
        if done:
            obs = env.reset()
    env.close()


if __name__ == "__main__":
    main()
```

A more full-featured random agent script is available in the examples dir:

```
python3 -m retro.examples.random_agent --game Airstriker-Genesis
```

It will print the current reward and will exit when the scenario is done. Note that it will throw an exception if no reward or scenario data is defined for that game. This script is useful to see if a scenario is properly set up and that the reward function isn't too generous.

Brute

There is a simple but effective reinforcement learning algorithm called "the Brute" from "Revisiting the Arcade Learning Environment" by Machado et al. which works on deterministic environments like Gym Retro games and is easy to implement. To run the example:

```
python3 -m retro.examples.brute --game Airstriker-Genesis
```

This algorithm works by building up a sequence of button presses that do well in the game, it doesn't look at the screen at all. It will print out the best reward seen so far while training.

PPO

Using "Proximal Policy Optimization" by Schulman et al., you can train an agent to play many of the games, though it takes awhile and is much faster with a GPU.

This example requires installing OpenAI Baselines. Once installed, you can run it:

```
python3 -m retro.examples.ppo --game Airstriker-Genesis
```

This will take awhile to train, but will print out progress as it goes. More information about PPO can be found in Spinning Up.

6.1.3 Integrations

What games have already been integrated? Note that this will display all defined environments, even ones for which ROMs are missing.

```
import retro
retro.data.list_games()
```

The actual integration data can be see in the Gym Retro Github repo.

6.1.4 Importing ROMs

If you have the correct ROMs on your computer (identified by the *rom.sha* file for each game integration), you can import them using the import script:

```
python3 -m retro.import /path/to/your/ROMs/directory/
```

This will copy all matching ROMs to their corresponding Gym Retro game integration directories.

Your ROMs must be in the *Supported ROM Types* list and must already have an integration. To add a ROM yourself, check out *Game Integration*.

Many ROMs should be available from the No-Intro Collection on Archive.org and the import script will search inside of zip files.

6.2 Python API

6.2.1 RetroEnv

The Python API consists primarily of `retro.make()`, `retro.RetroEnv`, and a few enums. The main function most users will want is `retro.make()`.

`retro.`**`make`**(*game*, *state=<State.DEFAULT: -1>*, *inttype=<Integrations.DEFAULT: <retro.data.DefaultIntegrations object>>*, ***kwargs*)

Create a Gym environment for the specified game

class retro.**RetroEnv**(*game*, *state=<State.DEFAULT: -1>*, *scenario=None*, *info=None*, *use_restricted_actions=<Actions.FILTERED: 1>*, *record=False*, *players=1*, *inttype=<Integrations.STABLE: 1>*, *obs_type=<Observations.IMAGE: 0>*)

Gym Retro environment class

Provides a Gym interface to classic video games

If you want to specify either the default state named in the game integration's *metadata.json* or specify that you want to start from the initial power on state of the console, you can use the *retro.State* enum:

class retro.**State**

Special values for setting the restart state of the environment. You can also specify a string that is the name of the `.state` file

DEFAULT = -1

Start the game at the default savestate from `metadata.json`

NONE = 0

Start the game at the power on screen for the emulator

6.2.2 Actions

There are a few possible action spaces included with *retro.RetroEnv*:

class retro.**Actions**

Different settings for the action space of the environment

ALL = 0

MultiBinary action space with no filtered actions

DISCRETE = 2

Discrete action space for filtered actions

FILTERED = 1

MultiBinary action space with invalid or not allowed actions filtered out

MULTI_DISCRETE = 3

MultiDiscete action space for filtered actions

You can also create your own action spaces derived from these. For an example, see discretizer.py. This file shows how to use `retro.Actions.Discrete` as well as how to make a custom wrapper that reduces the action space from `126` actions to `7`

6.2.3 Observations

The default observations are RGB images of the game, but you can view RAM values instead (often much smaller than the RGB images and also your agent can observate the game state more directly). If you want variable values, any variables defined in `data.json` will appear in the `info` dict after each step.

class retro.**Observations**

Different settings for the observation space of the environment

IMAGE = 0

Use RGB image observations

RAM = 1

Use RAM observations where you can see the memory of the game instead of the screen

6.2.4 Multiplayer Environments

A small number of games support multiplayer. To use this feature, pass `players=<n>` to *retro.RetroEnv*. Here is an example random agent that controls both paddles in `Pong-Atari2600`:

```
import retro

def main():
    env = retro.make(game='Pong-Atari2600', players=2)
    obs = env.reset()
    while True:
        # action_space will by MultiBinary(16) now instead of MultiBinary(8)
        # the bottom half of the actions will be for player 1 and the top half for␣
↪player 2
        obs, rew, done, info = env.step(env.action_space.sample())
        # rew will be a list of [player_1_rew, player_2_rew]
        # done and info will remain the same
        env.render()
        if done:
            obs = env.reset()
    env.close()


if __name__ == "__main__":
    main()
```

6.2.5 Replay files

Gym Retro can create .bk2 files which are recordings of an initial game state and a series of button presses. Because the emulators are deterministic, you will see the same output each time you play back this file. Because it only stores button presses, the file can be about 1000 times smaller than storing the full video.

In addition, if you wish to use the stored button presses for training, they may be useful. For example, there are replay files for each Sonic The Hedgehog level that were made available for the Gym Retro Contest.

You can create and view replay files using the *The Integration UI* (Game > Play Movie...). If you want to use replay files from Python, see the following sections.

Record

If you have an agent playing a game, you can record the gameplay to a `.bk2` file for later processing:

```
import retro

env = retro.make(game='Airstriker-Genesis', record='.')
env.reset()
while True:
    _obs, _rew, done, _info = env.step(env.action_space.sample())
    if done:
        break
```

Playback

Given a `.bk2` file you can load it in python and either play it back or use the actions for training.

```
import retro

movie = retro.Movie('Airstriker-Genesis-Level1-000000.bk2')
movie.step()

env = retro.make(
    game=movie.get_game(),
    state=None,
    # bk2s can contain any button presses, so allow everything
    use_restricted_actions=retro.Actions.ALL,
    players=movie.players,
)
env.initial_state = movie.get_state()
env.reset()

while movie.step():
    keys = []
    for p in range(movie.players):
        for i in range(env.num_buttons):
            keys.append(movie.get_key(i, p))
    env.step(keys)
```

Render to Video

This requires ffmpeg to be installed and writes the output to the directory that the input file is located in.

```
python3 -m retro.scripts.playback_movie Airstriker-Genesis-Level1-000000.bk2
```

6.3 Development

Adding new games can be done without recompiling Gym Retro, but if you need to work on the C++ code or make changes to the UI, you will want to compile Gym Retro from source.

6.3.1 Install Retro from source

Building Gym Retro requires at least either gcc 5 or clang 3.4.

Prerequisites

To build Gym Retro you must first install CMake. You can do this either through your package manager, download from the official site or `pip3 install cmake`. If you're using the official installer on Windows, make sure to tell CMake to add itself to the system PATH.

Mac prerequisites

Since LuaJIT does not work properly on macOS you must first install Lua 5.1 from homebrew:

```
brew install pkg-config lua@5.1
```

Windows prerequisites

If you are not on Windows, please skip to the next section. Otherwise, you will also need to download and install Git and MSYS2 x86_64. When you install git, choose to use Git from the Windows Command Prompt.

After you have installed msys2 open an MSYS2 MinGW 64-bit prompt (under Start > MSYS2 64bit) and run this command:

```
pacman -Sy make mingw-w64-x86_64-gcc
```

Once that's done, close the prompt and open a Git CMD prompt (under Start > Git) and run these commands. If you installed MSYS2 into an alternate directory please use that instead of C:msys64 in the command.

```
path %PATH%;C:\msys64\mingw64\bin;C:\msys64\usr\bin
set MSYSTEM=MINGW64
```

Then in the same prompt, without closing it first, continue with the steps in the next section. If you close the prompt you will need to rerun the last commands before you can rebuild.

Building

```
git clone https://github.com/openai/retro.git gym-retro
cd gym-retro
pip3 install -e .
```

6.3.2 Install Retro UI from source

First make sure you can install Retro from source, after that follow the instructions for your platform:

macOS

Note that for Mojave (10.14) you may need to install `/Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.14.pkg`

```
brew install pkg-config capnp lua@5.1 qt5
cmake . -DCMAKE_PREFIX_PATH=/usr/local/opt/qt -DBUILD_UI=ON -UPYLIB_DIRECTORY
make -j$(sysctl hw.ncpu | cut -d: -f2)
open "Gym Retro Integration.app"
```

Linux

```
sudo apt-get install capnproto libcapnp-dev libqt5opengl5-dev qtbase5-dev
cmake . -DBUILD_UI=ON -UPYLIB_DIRECTORY
make -j$(grep -c ^processor /proc/cpuinfo)
./gym-retro-integration
```

Windows

Building from source on Windows is currently difficult to configure. Docker containers for cross-compiling are available at openai/travis-build.

6.4 Game Integration

Integrating a game means taking a video game ROM file and setting it up as a reinforcement learning environment by defining 3 things:

- A starting state
- A reward function
- A done condition

Once integrated, you will be able to use the game through the Gym Retro Python API as a Gym environment.

If you are going to integrate a new game, you'll need a ROM for the correct system, see *Supported ROM Types* for a list.

6.4.1 Example Integration

This is a list of the integration files for the game Airstriker-Genesis.

`Level1.state`

This is a savestate from the beginning of the game, restarting the environment will put the agent at this point in the game.

`data.json`

This file defines the list of game-related variables that python can see based on their memory addresses in the games

```
{
  "info": {
    "gameover": {
      "address": 16712294,
      "type": ">u2"
    },
    "lives": {
      "address": 16712282,
      "type": ">u2"
    },
    "score": {
      "address": 16712270,
      "type": ">u4"
    }
  }
}
```

`scenario.json`

This file defines the reward function and done condition using the variables defined in *data.json*

```
{
  "done": {
    "condition": "all",
    "variables": {
```

(continues on next page)

(continued from previous page)

```
      "gameover": {
        "op": "equal",
        "reference": 1
      },
      "lives": {
        "op": "zero"
      }
    }
  },
  "reward": {
    "variables": {
      "score": {
        "reward": 1.0
      }
    }
  }
}
```

metadata.json

This file defines the default starting state if no state is specified by the user as well as some miscellaneous debugging information.

```
{
  "default_state": "Level1",
  "whitelist": {
    "data.json": [
      "suspicious type >u2 for lives"
    ]
  }
}
```

rom.md

This is the ROM file used for this game, with a few exceptions, ROM files are not included in Gym Retro, but will be in your local copy of Gym Retro after you import them.

rom.sha

This is the SHA1 hash of the *rom.md* file, used for importing ROMs.

```
a8be7b0ca850119b167f9644e6a4a22e983d61a4
```

These are all the files used in an integration. The next section will describe the files in more detail.

6.4.2 Integration Files

States

Emulation allows the entire state of a video game system to be stored to disk and restored. These files are specific to the emulator, but always end with *.state*. These are identical to the versions used in the standalone versions of the

emulators but gzipped.

Variable Locations `data.json`

Information about the inner workings of games are stored alongside the ROM in a file named `data.json`. This JSON file documents "ground truth" information about a game, including the locations and formats of variables in memory. These manifests are separated into sections, although only one section currently is defined:

The `info` section of the manifest lists game variables' memory addresses. Each entry in the `info` section consists of a key naming the memory address and the following values:

- `address`: The address into a RAM array of the first byte of the variable.
- `type`: A type descriptor for this variable. See the above addendum for the format of this value.

The following manifest shows an example of a game that has one variable, `score`, located at byte 128 that is 4 bytes wide in unsigned big endian format:

```
{
        "info": {
                "score": {
                        "address": 128,
                        "type": ">u4"
                }
        }
}
```

For more information on the possible variable types, see *Appendix: Types*.

Scenario `scenario.json`

Information pertaining to reward functions and done conditions can either be specified by manually overriding functions in *`retro.RetroEnv`* or can be done by writing a scenario file.

Scenario files contain information that is used to compute the reward function and done condition from variables defined in the information manifest. Each variable specified in the scenario file is multiplied by a `reward` value if positive and a `penalty` value if negative and then summed up to create the reward for that step. Similarly, states of these variables can be checked to see if the game is over. By default the scenario file will be loaded from `scenario.json`, but alternative scenario files can be specified in the *`retro.RetroEnv`* constructor.

Scenario files are again JSON and specified with the following sections:

The `reward` section used to calculate the reward function, and it split into the following subsections:

The `variables` subsection is used for defining how to calculate the reward function from the current state of memory. For each variable in the `variables` section, a value is calculated, multiplied by a coefficient, then added to the reward function for this step. How a value is extracted is specified by the `op`/`measurement`/`reference` values (see the addendum below on operations for the meanings of these). The default `measurement` is `delta`. There is no default `op`, and by default the value is passed through raw.

- `reward`: A coefficient multiplied by the value when the value is positive.
- `penalty`: A coefficient multiplied by the value when the value is negative.

A negative `penalty` would imply addition to the reward function instead of subtraction as the value to be multiplied by the coefficient is negative.

The `time` subsection is used for creating rewards based off of how many steps are taken. Two values can be specified:

- `reward`: A value to be added to the reward function every step.

- `penalty`: A value to be subtracted from the reward function every step.

The `done` section is used to calculate if the end of a game has been reached. At the top level the following property is available:

- `condition`: Specifies how the `done` conditions should be combined - `any`: Any of the conditions in the `done` section is fulfilled. This is the default. - `all`: All of the conditions in the `done` section are fulfilled.

Currently it has one subsection:

The `variables` subsection specifies how to calculate the done condition from the current state of memory. Each variable in the `variables` subsection is extracted per the `op`/`measurement`/`reference` values (see the addendum below on operations for the meanings of these). The default `measurement` is `absolute`. There is no default `op`, and by default the value is ignored.

For more information on the conditions that can be defined, see *Appendix: Operations*.

6.4.3 The Integration UI

The integration UI helps you easily find variables and see what is going on with the reward function. You can download the compiled UI package for your platform here:

- Windows Integration UI
- Mac Integration UI

Integrating a new ROM

1. Open the Gym Retro Integration UI
2. Load a new game — `Command-Shift-O` on Mac
3. Select the ROM of the game you'd like to integrate in the menu
4. Name the game
5. The game will open. To see what keys correspond to what controls in-game, go to Window > Control
6. Using the available controls, select a level, option, mode, character, etc. and take note of these options
7. When you are finally at the first playable moment of the game, pause the game (in the integrator, not within the actual game) (`Command-P`), and save the state (`Command-S`). This moment can be hard to find, and you might have to go back through and restart the game (`Command-R`) to find and save that exact state.
8. Save the state — include the options you chose in the previous menus — e.g. `SailorMoon.QueenBerylsCastle.Easy.Level1.state`

For Gym Retro integrations, a few notes about ROMs:

- We have preferred the USA version of ROMs, denoted by one of `(USA)`, `(USA, Europe)`, `(Japan, USA)`, etc
- If the ROM has a `.bin` extension, rename it to have the correct extension for that system listed in *Supported ROM Types*
- Use the Gym Retro Integration application and select the Integrate option from the File menu to begin working on integrating it

6.4.4 Supported ROM Types

ROM files contain the game itself. Each system has a unique file extension to denote which system a given ROM runs on:

- `.md`: Sega Genesis (also known as Mega Drive)
- `.sfc`: Super Nintendo Entertainment System (also known as Super Famicom)
- `.nes`: Nintendo Entertainment System (also known as Famicom)
- `.a26`: Atari 2600
- `.gb`: Nintendo Game Boy
- `.gba`: Nintendo Game Boy Advance
- `.gbc`: Nintendo Game Boy Color
- `.gg`: Sega Game Gear
- `.pce`: NEC TurboGrafx-16 (also known as PC Engine)
- `.sms`: Sega Master System

Sometimes ROMs from these systems use different extensions, e.g. `.gen` for Genesis, `.bin` for Atari, etc. Please rename the ROMs to use the aforementioned extensions in these cases.

6.4.5 Integrating a Game

To integrate a game you need to define a done condition and a reward function. The done condition lets Gym Retro know when to end a game session, while the reward function provides a simple numeric goal for machine learning agents to maximize.

To define these, you find variables from the game's memory, such as the player's current score and lives remaining, and use those to create the done condition and reward function. An example done condition is when the `lives` variable is equal to `0`, an example reward function is the change in the `score` variable.

Note: if the game requires that you hit the `Start` button to play, for instance after dying, then you need to modify the scenario file to allow this as `Start` is disallowed by default. See the `actions` key in KidChameleon-Genesis for an example of this.

Done Condition

This is usually the easier of the two. The best done condition to use is the Game Over or Continue screen after you run out of lives. For some games this is when you have zero lives left, for some `-1` lives, for others, it can be pretty hard.

It's better to have a simple and reliable but slightly incorrect done condition (e.g. ending the game when you still have 1 life left because it's hard to detect the 0 life case) than to have a done condition that is unreliable, such as a `gameover` variable that detects when the gameover screen is present most of the time but also incorrectly fires when switching levels.

If you create a `gameover` variable, make sure to test it with a replay that plays multiple levels in a row to make sure it doesn't fire accidentally.

Reward Function

Reinforcement learning agents try to maximize the reward function. The ideal reward function would be that you get 1 point for beating the game. There's no way to maximize that besides beating the game.

That reward is impractical though, because existing reinforcement learning algorithms are unable to make progress with a reward that is so hard to get. Instead we can specify some easier to get reward that, if you maximize it, should result in beating the game.

If the game has a score, this is often a good choice. In some games however, you can get as much score as you want by standing in one place and attacking the same enemy over and over as it respawns. Because that is so different from beating the game, it's best to have an alternative reward, though these are often very game specific.

Be careful with non-score variables though, they can be tricky to get right, make sure to play multiple levels using the reward to see if it makes sense.

Provided you use the score, define a `score` variable and set the reward such that the reward the agent receives matches the score displayed on the screen, make sure to check that you're not off by a factor of 10 or 100 by comparing to the `Cumulative` value displayed in the `Scenario Information` pane of the UI.

Finding Variables

It's best to keep a consistent pattern for the different types of variables you might add to a game's `data.json` file. Here are some tips:

1. It's pretty common for multiple different variables to group themselves together. When narrowing down the search for a particular variable, look at nearby memory addresses if you suspect you have a similar but incorrect variable (for instance you found the high score variable but are looking for the score variable).
2. Score occasionally is stored in individual locations — e.g. if the score displayed is `123400`, 1, 2, 3, 4, 0, 0 all will update separately. If the score is broken into multiple variables, make sure you have penalties set for the individual digits (such as BOB-Snes). A number of games will update the score value across multiple frames, in this case you will need a lua script to correct the reward, such as 1942-Nes.
3. Check for uncommon lengths of 3, 5, etc. Games don't always store score in nice neat lengths of multiples of 2, and making sure the variable is the appropriate length is key — if you go too short, then no progress over a certain score is tracked, if you go too long, then the score can suddenly jump between levels, etc. If you can't decide between two possible lengths, the shorter length is the safer bet.
4. Score variable doesn't always include the 0s at the end of the game — while the screen might say `2400`, the score variable might only store `24`. So you will need to multiply by 100 in this case.
5. It's very uncommon, but occasionally, scores can be transposed by a value of 1 — e.g., while the screen says `123456`, the variable is `012345`. Some of these scores start at -1 rather than 0. This can be fixed with lua.
6. It's very uncommon, but some games track health symbolically rather than with one set #. For example, the starting health bar could be represented by `9999999`, which displays as a full health bar, but becomes `99999` after losing two health units.
7. In defining a game over variable, look for a binary value that switches between 0 and 1 – 0 when the game is in play, 1 when the game is over. And make sure to test it by playing a few consecutive levels.

Once you've found a variable, making sure the address and type are correct is important for avoiding issues later. One of the best ways to do this is to change the value in memory, then change it in the game and make sure it updates correctly.

For instance if you have a variable called "score" and you want to see if it is `>d2` or `>d4`, set the type to `>d4` and set the value to the maximum for `>d2`, 9999, and then increase the score by playing the game. If the score increases by 1, and the value in the memory viewer is 10000 and the value in the game is 10000, then `>d4` is correct. If the value in the memory viewer or game is 0 or 9999, then it's likely that `>d2` is the correct type or that the address is wrong. You may also want to check if >d3 is the correct type by changing the score to 999999 and playing for a bit.

You can also check to see if the data type is correct by watching how it increments and decrements in the search window as you play the game. For example, if the value of the variable jumps from `0` to `255`, it's likely that this is a signed value (represented by `i`) – unsigned values (represented by `u`) are either positive or zero.

When you search for a variable, different formats at the same location will appear next to each other in the search window. For example, at address `16769105`, you might see `>u2`, `>i2`, as well as `>d2` return as search results. Play the game for a little bit, and you might notice that one of the search results increments/decrements in uneven or unusual ways in relation to the other search results at the same address.

eg: `|u1` at `7e094d` goes from `144` -> `137`, `|d1` at `7e094d` (same address) decrements from `90` -> `89`, it's probably `|d1`

If you update the value of a variable but it doesn't have any effect on the game, it's likely that you've found a copy of the variable, not the correct address. An example would be a lives variable, but setting it to some higher value and then dying in the game reveals that you didn't actually increase the number of lives. It's often the case that you have to change the value in the game to get things to update (such as losing a life in the previous example).

Ideally you can find the original since it's more likely to be correct, so if you can, find a variable that when updated, updates the corresponding value in the game. The most common source of this is a high score variable which will have the same value as the normal score variable, but updating it will have no effect on your score.

Common Errors

- Wrong type for variable: if your score variable is actually `>d2` and you put `>d4`, you may not notice until you get to some later level and the memory address next to the score is used for something, suddenly giving you a very large score.
- Incorrect done condition: it might be that if you run out of time or die in some unusual way that the done condition is not detected correctly. Make sure to test unusual ways of ending the game, and make sure that your done condition doesn't fire upon completing a level (unless it's the final level of the game). If you're able to hit continue after dying, make sure that the game ends before the agent can hit continue.
- Score is used as reward, but it's different from the score displayed in the game: this could happen if you forgot a factor of 10 in the reward, or if you're calculating the score based on some other variables (e.g. the upper and lower digits of the score, or some variable like `number of enemies killed * 100`) and there is a bug. If you play the game for awhile and the reward diverges slightly from the in-game score, it's possible that the score digits are not always updated at the same time. In this case, you can use the change in maximum score as the reward, see GuardianLegend-Nes for an example of this.

6.4.6 Using a Custom Integration from Python

Once you have created an integration, you can put it in a folder called `custom_integrations` and tell `retro` about your custom integration using the `add_custom_path` function:

```
import retro
import os

SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))


def main():
        retro.data.Integrations.add_custom_path(
                os.path.join(SCRIPT_DIR, "custom_integrations")
        )
        print("FakeGame-Nes" in retro.data.list_games(inttype=retro.data.Integrations.
↪ALL))
        env = retro.make("FakeGame-Nes", inttype=retro.data.Integrations.ALL)
        print(env)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
        main()
```

This lets you use your integration without having to add it to `retro` directly.

6.4.7 Appendix: Types

The types consist of three parts, in order:

- Endianness
- Format
- Bytes

Endianness refers to the order of the bytes in memory. For example, take the hex string `0x01020304`, which can be stored many ways:

- Big endian: `0x01 0x02 0x03 0x04`
- Little endian: `0x04 0x03 0x02 0x01`
- Middle endian (big outside/little inside): `0x02 0x01 0x04 0x03`
- Middle endian (little outside/big inside): `0x03 0x04 0x01 0x02`

The following sigils correspond to the endiannesses:

- <: Little
- >: Big
- ><: Middle (big/little)
- <>: Middle (little/big)
- =: Native (little on most computers)
- >=: Middle (big/native)
- <=: Middle (little/native)
- `|`: Don't care (only useful for single-byte values)

NB: Middle endian is very rare, but some systems store 16-bit values in native endian and 32-bit values as two 16-bit values in big endian order. One such example is the emulator Genesis Plus GX. Thus, on a big endian system the format appears to be `=u4` (aka `>u4`) when it appears as `>=u4` on little endian systems. As such some data may require manual grooming.

Format refers to how in memory a value is stored. For example, take the hex byte `0x81`. It could mean three things in decimal:

- Unsigned: 129
- Signed: -127
- Binary-coded decimal: 81
- Low-nybble Binary-coded decimal: 1

NB: The nybbles `0xA` - `0xF` cannot occur in binary-coded decimal.

The following characters correspond to formats:

- `i`: Signed

- `u`: Unsigned
- `d`: Binary-coded Decimal
- `n`: Low-nybble Binary-coded Decimal

Finally, the last piece refers to how many bytes a value occupies in memory. Ideally, this should be a power of two, e.g. 1, 2, 4, 8, etc., however non-power of two values are used by some games (e.g. the score in Super Mario Bros. is 6 bytes long), so non-power of two variables are supported.

NB: Native endian and middle endian don't work with non-power of two sizes or sizes less than 4 bytes. Currently only 4-byte middle endian is properly supported.

Some examples follow:

- `<u2`: Little endian two-byte unsigned value (i.e. `0x0102` -> `0x02 0x01`)
- `<>u4`: Middle endian (little/big) four-byte unsigned value (i.e. `0x01020304` -> `0x03 0x04 0x01 0x02`)
- `>d2`: Big endian two-byte binary-coded decimal value (i.e. `1234` -> `0x12 0x34`)
- `|u1`: Single unsigned byte
- `<u3`: Non-power of two bytes (i.e. `0x010203` -> `0x03 0x02 0x1`)
- =n2: Native endian two-byte low-nybble binary-coded decimal value (i.e. `12` -> `0x01 0x02` on Intel and most ARM CPUs, `0x02 0x01` on PowerPC CPUs)

Some non-examples:

- `|i2`: Valid but not recommended: Two signed bytes, order undefined
- `<u1`: Valid but not recommended: One byte has no order
- `?u4`: Invalid: undefined endianness
- `>q2`: Invalid: undefined format
- `=i0`: Invalid: zero bytes
- `><u3`: Invalid: Non-power of two middle endian bytes
- `<=u2`: Invalid: Middle endian does not make sense for two byte values

6.4.8 Appendix: Operations

Games can store information in memory in many various ways, and as such the specific information needed can vary in form too. The basic premise is that once a raw value is extracted from memory an operation may be defined to transform it to a useful form. Furthermore, we may want raw values in a given step or the deltas between two steps. Thus three properties are defined:

- `measurement`: The method used for extracting the raw value. May be `absolute` for the current value and `delta` for the difference between the current and previous value. The default varies based on context.
- `op`: The specific operation to apply to this value. Valid operations are defined below.
- `reference`: The reference value for an operation, if needed.

The following operations are defined:

- `nonzero`: Returns 0 if the value is 0, 1 otherwise.
- `zero`: Returns 1 if the value is 0, 0 otherwise.
- `positive`: Returns 1 if the value is positive, 0 otherwise.
- `negative`: Returns 1 if the value is negative, 0 otherwise.

- `sign`: Returns 1 if the value is positive, -1 if the value is negative, 0 otherwise.
- `equal`: Returns 1 if the value is equal to the `reference` value, 0 otherwise.
- `not-equal`: Returns 1 if the value is not equal to the `reference` value, 0 otherwise.
- `less-than`: Returns 1 if the value is less than the `reference` value, 0 otherwise.
- `greater-than`: Returns 1 if the value is greater than the `reference` value, 0 otherwise.
- `less-or-equal`: Returns 1 if the value is less than or equal to the `reference` value, 0 otherwise.
- `greater-or-equal`: Returns 1 if the value is greater than or equal to the `reference` value, 0 otherwise.

Index