1 Import and Start the Demo VM

1. Install VirtualBox using the provided .exe or .dmg installer. (Linux users: as distributions vary, please use your package manager.)

   You should also be able to import the VM into VMware, but we have not tested this.

2. From the File menu, select Import Appliance and browse to the .ova file containing the VM and click Next, then Import.

3. Click the newly-imported VM in the list of VMs in the left panel, then click the Start button on the toolbar.

2 Tune Mario with Defaults

We will run the Mario example included in the OpenTuner repository, in which OpenTuner plays the first level of Super Mario Bros., minimizing the time required to complete the level. (For copyright reasons, the Super Mario Bros. ROM file is not included in the repository, but it is included in the VM.)

1. Log in as user `demo` with password `demo`.

2. Open a terminal by clicking the taskbar button in the lower-left corner of the screen, select System Tools, then click UXTerm.

3. `cd opentuner/examples/mario`

4. Run the Mario example with `./mario.py --technique=PSO_GA_Bandit --parallelism=1 --stop-after=300 --headful --representation=DurationRepresentation --fitness-function=ProgressTimesAverageSpeed`. You will see one instance of the NES emulator running the configurations OpenTuner generated. In the terminal, OpenTuner will print whether each configuration won or lost, the number of pixels Mario moved to the right, and the total number of frames elapsed during the trial.

   The first three options are general OpenTuner options:

   - `--technique` selects which technique OpenTuner will use to generate new configurations. This can be a single technique or a bandit composing multiple techniques. Use the `--list-techniques` option to see all available techniques. `PureRandom` is random search.
   - `--parallelism` specifies how many configurations OpenTuner will test in parallel. For autotuning program performance, this should usually be 1 to prevent interference between trials. Multiple trials in the Mario example do not interfere, so this value can be increased.
   - `--stop-after` specifies the duration of the tuning run, in seconds. The tuning run can also be ended early by sending Ctrl-C to OpenTuner.

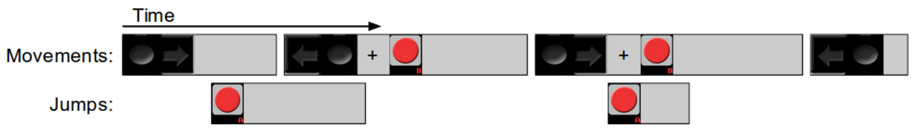

Figure 1: Interpreting the DurationRepresentation parameters.

The remaining options are specific to the Mario example. `--headful` specifies that the emulators should render the game; if this option is absent, the emulators will be hidden (saving some CPU cycles). `--representation` and `--fitness-function` select the representation and fitness function to use and will be detailed in the upcoming sections.

3 Experiment with Representations

The Mario example has support for pluggable configuration representations (sets of parameters). The NES emulator takes as input the set of buttons being pressed on each video frame. In `mario.py`, the `Representation` class has one method that creates the parameters that OpenTuner can use and another method that interprets these parameters' values in terms of button presses. You can experiment with other representations by creating a new representation subclass and implementing these methods.

The `--representation` option specifies the name of the representation class to use.

`mario.py` already contains two representation implementations:

- `NaiveRepresentation`, which includes one boolean parameter for each button for each frame, up to 12000 frames. The parameters are interpreted in the obvious way: the button is pressed on a given frame if the corresponding parameter is true.

- `DurationRepresentation`, which encodes the button presses as (action, duration) pairs, one for movement and the other for jumps. The parameters are interpreted by performing the first movement action for the first duration, then the second action for the second duration and so on, and similarly placing the jumps and their durations. (See the figure.)

Here are some ideas to try:

- Create a `DurationRepresentation` variant that handles jumps as part of movement, instead of a separate group of parameter pairs.

- Create a representation that always runs to the right (only tuning jumps).

4 Experiment with Fitness Functions

The Mario example also supports pluggable fitness functions, which tell OpenTuner how successful a configuration was. In `mario.py`, the `FitnessFunction` class has one method that calculates the fitness based on whether the configuration won (Mario reached the end of the level), how many pixels Mario moved to the right, and how many frames elapsed before Mario won or died. You can implement a new fitness function by subclassing `FitnessFunction` and implementing this method.

The `--fitness-function` option specifies the name of the fitness function class to use.

`mario.py` already contains three fitness functions:

- `Progress`, which simply returns the number of pixels Mario moved to the right.
- `ProgressPlusTimeRemaining`, which returns the number of pixels Mario moved to the right, plus the number of frames remaining in the time limit if Mario won. This function rewards configurations that win faster, but the all-or-nothing bonus results in a large discontinuity in the fitness function at the win point.
- `ProgressTimesAverageSpeed`, which returns the number of pixels Mario moved to the right multiplied by Mario's average speed (pixels per frame). This fitness function rewards both progress and speed over the whole tuning run, rather than only rewarding speed for winning configurations.

Here are some ideas to try:

- Implement fitness functions that weight progress or speed logarithmically or exponentially.
- Implement a fitness function that minimizes time without consideration of progress. OpenTuner should search for a configuration that kills Mario as quickly as possible (probably by running directly into the first enemy).