PyGame Learning Environment Documentation

Release 0.1.dev1

Norman Tasfi

May 09, 2016

Contents

PyGame Learning Environment (PLE) is a learning environment, mimicking the Arcade Learning Environment interface, allowing a quick start to Reinforcement Learning in Python. The goal of PLE is allow practitioners to focus design of models and experiments instead of environment design.

PLE has only been tested with **Python 2.7.6**.

CHAPTER 1

User Guide

The PLE user guide below explains the different components inside of the library. It covers how to train a reinforcement learning agent, the environments structure and function, and the required methods if one wishes to add games to library.

1.1 Home

1.1.1 Installation

PLE requires the following libraries to be installed:

- numpy
- pillow
- pygame

PyGame can be installed using this tutorial (Ubuntu). For mac you can use these following instructions;

```
brew install sdl sdl_ttf sdl_image sdl_mixer portmidi  # brew or use equivalent means
conda install -c https://conda.binstar.org/quasiben pygame  # using Anaconda
```

To install PLE first clone the repo:

```
git clone https://github.com/ntasfi/PyGame-Learning-Environment
```

Then use the `cd` command to enter the `PyGame-Learning-Environment` directory and run the command:

```
sudo pip install -e .
```

This will install PLE as an editable library with pip.

1.1.2 Quickstart

PLE allows agents to train against games through a standard model supplied by `ple.PLE`, which interacts and manipulates games on behalf of your agent. PLE mimics the Arcade Learning Environment (ALE) interface as closely as possible. This means projects using the ALE interface can easily be adjusted to use PLE with minimal effort.

If you do not wish to perform such modifications you can write your own code that interacts with PLE or use libraries with PLE support such as General Deep Q RL.

Here is an example of having an agent run against `FlappyBird`.

```
from ple.games.flappybird import FlappyBird
from ple import PLE


game = FlappyBird()
p = PLE(game, fps=30, display_screen=True)
agent = myAgentHere(allowed_actions=p.getActionSet())

p.init()
reward = 0.0

for i in range(nb_frames):
   if p.game_over():
           p.reset_game()

   observation = p.getScreenRGB()
   action = agent.pickAction(reward, observation)
   reward = p.act(action)
```

1.2 Tutorials

1.2.1 Wrapping and Adding Games

Adding or wrapping games to work with PLE is relatively easy. You must implement a few methods, explained below, to have a game be useable with PLE. We will walk through an implementation of the Catcher game inspired by Eder Santana to examine the required methods for games. As we want to focus on the important aspects related to the game interface we will ignore game specific code.

Note: The full code is not included in each method. The full implementation, which implements scaling based on screen dimensions, is found here.

Catcher is a simple game where the agent must catch 'fruit' dropped at random from the top of the screen with the 'paddle' controlled by the agent.

The main component of the game is enclosed in one class that inherits from `base.Game`:

```
from ple.games import base
from pygame.constants import K_a, K_d

class Catcher(base.Game):

        def __init__(self, width=48, height=48):

                actions = {
                        "left": K_a,
                        "right": K_d
                }

                base.Game.__init__(self, width, height, actions=actions)

                #game specific
                self.lives = 0
```

The game must inherit from `base.Game` as it sets attributes and methods used by PLE to control game flow, scoring and other functions.

The crucial porition in the __init__ method is to call the parent class __init__ and pass the width, height and valid actions the game responds too.

Next we cover four required methods: `init`, `getScore`, `game_over`, and `step`. These methods are all required to interact with our game.

The code below is within our `Catcher` class and has the class definition repeated for clarity:

```
class Catcher(base.Game):

        def init(self):
                self.score = 0

                #game specific
                self.lives = 3

        def getScore(self):
                return self.score

        def game_over(self):
                return self.lives == 0

        def step(self, dt):
                #move players
                #check hits
                #adjust scores
                #remove lives
```

The `init` method sets the game to a clean state. The minimum this method must do is to reset the `self.score` attribute of the game. It is also strongly recommended this method perform other game specific functions such as player position and clearing the screen. This is important as the game might still be in a terminal state if the player and object positions are not reset; which would result in endless resetting of the environment.

`getScore` returns the current score of the agent. You are free to pull information from the game to decide on a score, such as the number of lives left etc. or you can simply return the `self.score` attribute.

`game_over` must return True if the game has hit a terminal state. This depends greatly on game. In this case the agent loses a life for each fruit it fails to catch and causes the game to end if it hits 0.

`step` method is responsible for the main logic of the game. It is called everytime our agent performs an action on the game environment. `step` performs a step in game time equal to `dt`. `dt` is required to allow the game to run at different frame rates such that the movement speeds of objects are scaled by elapsed time. With that said the game can be locked to a specific frame rate, by setting `self.allowed_fps`, and written such that `step` moves game objects at rates suitable for the locked frame rate. The function signature always expects `dt` to be passed, the game logic does not have to use it though.

Thats it! You only need a handful of methods defined to be able to interface your game with PLE. It is suggested to look through the different games inside of the games folder.

1.2.2 Non-Visual State Representation

Sometimes it is useful to have non-visual state representations of games. This could be to try reduced state space, augment visual input, or for troubleshooting purposes. The majority of current games in PLE support non-visual state representations. To use these representations instead of visual inputs one needs to inspect the state structure given in the documentation. You are free to select sub-poritions ofthe state as agent input.

Lets setup an agent to use a non-visual state representation of `Pong`.

First start by examining the values `Pong` will return from the `getGameState()` method:

```
def getGameState(self):
    #other code above...

    state = {
        "player_y": self.agentPlayer.pos.y,
        "player_velocity": self.agentPlayer.vel.y,
        "cpu_y": self.cpuPlayer.pos.y,
        "ball_x": self.ball.pos.x,
        "ball_y": self.ball.pos.y,
        "ball_velocity_x": self.ball.pos.x,
        "ball_velocity_y": self.ball.pos.y
    }

    return state
```

We see that `getGameState()` of Pong returns several values each time it is called. Using the returned dictonary we can create a numpy vector representating our state.

This can be accomplished in the following ways:

```
#easiest
my_state = np.array([ state.values() ])

#by-hand
my_state = np.array([ state["player"]["x"], state["player"]["velocity"], ... ])
```

You have control over which values you want to include in the state vector. Training an agent would look like this:

```
from ple.games.pong import Pong
from ple import PLE
import numpy as np

def process_state(state):
        return np.array([ state.values() ])

game = Pong()
p = PLE(game, display_screen=True, state_preprocessor=process_state)
agent = myAgentHere(input_shape=p.getGameStateDims(), allowed_actions=p.getActionSet())

p.init()
nb_frames = 10000
reward = 0.0
for i in range(nb_frames):
   if p.game_over():
           p.reset_game()

   state = p.getGameState()
   action = agent.pickAction(reward, state)
   reward = p.act(action)
```

To make this work a state processor must be supplied to PLE's `state_preprocessor` initialization method. This function will be called each time we request the games state. We can also let our agent know the dimensions of the vector to expect from PLE. In the main loop just simply replace the call to `getScreenGrayScale` with `getGameState`.

Be aware different games will have different dictonary structures. The majority of games will return back a flat dictory structure but others will have lists inside of them. In particular games with variable objects to track, such as the number of segments in snake, require usage of lists within the dictonary.

```
state = {
    "snake_head_x": self.player.head.pos.x,
    "snake_head_y": self.player.head.pos.y,
    "food_x": self.food.pos.x,
    "food_y": self.food.pos.y,
    "snake_body": []
}
```

The `"snake_body"` field contains a dynamic number of values. It must be taken into consideration when creating your state preprocessor.

1.3 Available Games

1.3.1 Catcher

In Catcher the agent must catch falling fruit with its paddle.

Valid Actions

Left and right control the direction of the paddle. The paddle has a little velocity added to it to allow smooth movements.

Terminal states (game_over)

The game is over when the agent lose the number of lives set by init_lives parmater.

Rewards

The agent receives a positive reward, of +1, for each successful fruit catch, while it loses a point, -1, if the fruit is not caught.

class `ple.games.catcher.`**`Catcher`**(*width=64*, *height=64*, *init_lives=3*)

Based on Eder Santana's game idea.

Parameters **width** : int

Screen width.

height : int

Screen height, recommended to be same dimension as width.

init_lives : int (default: 3)

The number lives the agent has.

`getGameState`()

Gets a non-visual state representation of the game.

Returns dict

- player x position.
- players velocity.
- fruits x position.

- fruits y position.

See code for structure.

1.3.2 Monster Kong

A spinoff of the original Donkey Kong game. Objective of the game is to avoid fireballs while collecting coins and rescuing the princess. An additional monster is added each time the princess is rescued.

Valid Actions

Use w, a, s, d and space keys to move around player around.

Terminal states (game_over)

The game is over when the player hits three fireballs. Touching a monster does not kill cause the agent to lose lives.

Rewards

The player gains +5 for collecting a coin while losing a life and receiving a reward of -25 for hitting a fireball. The player gains +50 points for rescusing a princess.

Note: Images were sourced from various authors. You can find the respective artist listed in the assets directory folder of the game.

1.3.3 FlappyBird

Flappybird is a side-scrolling game where the agent must successfully nagivate through gaps between pipes.

FPS Restrictions

This game is restricted to 30fps as the physics at higher and lower framerates feels slightly off. You can remove this by setting the *allowed_fps* parameter to *None*.

Valid Actions

Up causes the bird to accelerate upwards.

Terminal states (game_over)

If the bird makes contact with the ground, pipes or goes above the top of the screen the game is over.

Rewards

For each pipe it passes through it gains a positive reward of +1. Each time a terminal state is reached it receives a negative reward of -1.

class ple.games.flappybird.**FlappyBird**(*width=288*, *height=512*, *pipe_gap=100*)

Used physics values from sourabhv's clone.

Parameters **width** : int (default: 288)

Screen width. Consistent gameplay is not promised for different widths or heights, therefore the width and height should not be altered.

height : inti (default: 512)

Screen height.

pipe_gap : int (default: 100)

The gap in pixels left between the top and bottom pipes.

getGameState()

Gets a non-visual state representation of the game.

Returns dict

- player y position.
- players velocity.
- next pipe distance to player
- next pipe top y position
- next pipe bottom y position
- next next pipe distance to player
- next next pipe top y position
- next next pipe bottom y position

See code for structure.

1.3.4 Pixelcopter

Pixelcopter is a side-scrolling game where the agent must successfully nagivate through a cavern. This is a clone of the popular helicopter game but where the player is a humble pixel.

Valid Actions

Up which causes the pixel to accelerate upwards.

Terminal states (game_over)

If the pixel makes contact with anything green the game is over.

Rewards

For each vertical block it passes through it gains a positive reward of +1. Each time a terminal state reached it receives a negative reward of -1.

class ple.games.pixelcopter.**Pixelcopter**(*width=48*, *height=48*)

Parameters **width** : int

Screen width.

height : int

Screen height, recommended to be same dimension as width.

getGameState()

Gets a non-visual state representation of the game.

Returns dict

- player y position.
- player velocity.
- player distance to floor.
- player distance to ceiling.
- next block x distance to player.
- next blocks top y location,
- next blocks bottom y location.

See code for structure.

1.3.5 Pong

Pong simulates 2D table tennis. The agent controls an in-game paddle which is used to hit the ball back to the other side.

The agent controls the left paddle while the CPU controls the right paddle.

Valid Actions

Up and down control the direction of the paddle. The paddle has a little velocity added to it to allow smooth movements.

Terminal states (game_over)

The game is over if either the agent or CPU reach the number of points set by MAX_SCORE.

Rewards

The agent receives a positive reward, of +1, for each successful ball placed behind the opponents paddle, while it loses a point, -1, if the ball goes behind its paddle.

class ple.games.pong.**Pong**(*width=64*, *height=48*, *MAX_SCORE=11*)

Loosely based on code from marti1125's pong game.

Parameters **width** : int

Screen width.

height : int

Screen height, recommended to be same dimension as width.

MAX_SCORE : int (default: 11)

The max number of points the agent or cpu need to score to cause a terminal state.

getGameState()

Gets a non-visual state representation of the game.

Returns dict

- player y position.
- players velocity.
- cpu y position.
- ball x position.
- ball y position.
- ball x velocity.
- ball y velocity.

See code for structure.

1.3.6 PuckWorld

In PuckWorld the agent, a blue circle, must navigate towards the green dot while avoiding the larger red puck.

The green dot randomly moves around the screen while the red puck slowly follows the agent.

Valid Actions

Up, down, left and right apply thrusters to the agent. It adds velocity to the agent which decays over time.

Terminal states (game_over)

None. This is a continuous game.

Rewards

The agent is rewarded based on its distance to the green dot, where lower values are good. If the agent is within the large red radius it receives a negative reward. The negative reward is proportional to the agents distance from the pucks center.

class ple.games.puckworld.**PuckWorld**(*width=64*, *height=64*)

Based Karpthy's PuckWorld in REINFORCEjs.

Parameters **width** : int

Screen width.

height : int

Screen height, recommended to be same dimension as width.

getGameState()
Gets a non-visual state representation of the game.

Returns dict

- player x position.
- player y position.
- players x velocity.
- players y velocity.
- good creep x position.
- good creep y position.
- bad creep x position.
- bad creep y position.

See code for structure.

1.3.7 RaycastMaze

In RaycastMaze the agent must navigate a 3D environment searching for the exit denoted with a bright red square.

It is possible to increase the map size by 1 each time it successfully solves the maze. As seen below.

Example

```
>>> #init and setup etc.
>>> while True:
>>>   if game.game_over():
>>>     game.map_size += 1
>>>   game.step(dt) #assume dt is given
```

Not valid code above.

Valid Actions

Forwards, backwards, turn left and turn right.

Terminal states (game_over)

When the agent is a short distance, nearly touching the red square, the game is considered over.

Rewards

Currently it receives a postive reward of +1 when it finds the red block.

class ple.games.raycastmaze.**RaycastMaze**(*init_pos=(1, 1)*, *resolution=1*, *move_speed=20*, *turn_speed=13*, *map_size=10*, *height=48*, *width=48*)

Parameters **init_pos** : tuple of int (default: (1,1))

The position the player starts on in the grid. The grid is zero indexed.

resolution : int (default: 1)

This instructs the Raycast engine on how many vertical lines to use when drawing the screen. The number is equal to the width / resolution.

move_speed : int (default: 20)

How fast the agent moves forwards or backwards.

turn_speed : int (default: 13)

The speed at which the agent turns left or right.

map_size : int (default: 10)

The size of the maze that is generated. Must be greater then 5. Can be incremented to increase difficulty by adjusting the attribute between game resets.

width : int (default: 48)

Screen width.

height : int (default: 48)

Screen height, recommended to be same dimension as width.

getGameState()

Returns None

Does not have a non-visual representation of game state. Would be possible to return the location of the maze end.

1.3.8 Snake

Snake is a game where the agent must maneuver a line which grows in length each time food is touched by the head of the segment. The line follows the previous paths taken which eventually become obstacles for the agent to avoid.

The food is randomly spawned inside of the valid window while checking it does not make contact with the snake body.

Valid Actions

Up, down, left, and right. It cannot turn back on itself. Eg. if its moving downwards it cannot move up.

Terminal states (game_over)

If the head of the snake comes in contact with any of the walls or its own body (can occur after only 7 segments) the game is over.

Rewards

It recieves a positive reward, +1, for each red square the head comes in contact with. While getting -1 for each terminal state it reaches.

class ple.games.snake.**Snake**(*width=64*, *height=64*, *init_length=3*)

Parameters **width** : int

Screen width.

height : int

Screen height, recommended to be same dimension as width.

init_length : int (default: 3)

The starting number of segments the snake has. Do not set below 3 segments. Has issues with hitbox detection with the body for lower values.

getGameState()

Returns dict

- snake head x position.
- snake head y position.
- food x position.
- food y position.
- distance from head to each snake segment.

See code for structure.

1.3.9 WaterWorld

In WaterWorld the agent, a blue circle, must navigate around the world capturing green circles while avoiding red ones.

After capture a circle it will respawn in a random location as either red or green. The game is over if all the green circles have been captured.

Valid Actions

Up, down, left and right apply thrusters to the agent. It adds velocity to the agent which decays over time.

Terminal states (game_over)

The game ends when all the green circles have been captured by the agent.

Rewards

For each green circle captured the agent receives a positive reward of +1; while hitting a red circle causes a negative reward of -1.

class ple.games.waterworld.**WaterWorld**(*width=48*, *height=48*, *num_creeps=3*)

Based Karpthy's WaterWorld in REINFORCEjs.

Parameters **width** : int

Screen width.

height : int

Screen height, recommended to be same dimension as width.

num_creeps : int (default: 3)

The number of creeps on the screen at once.

getGameState()

Returns dict

- player x position.
- player y position.
- player x velocity.
- player y velocity.
- player distance to each creep

CHAPTER 2

API Reference

Information for specific classes and methods.

2.1 ple.PLE

class ple.**PLE**(*game*, *fps=30*, *frame_skip=1*, *num_steps=1*, *reward_values={}*, *force_fps=True*, *display_screen=False*, *add_noop_action=True*, *NOOP=K_F15*, *state_preprocessor=None*, *rng=24*)

Main wrapper that interacts with games. Provides a similar interface to Arcade Learning Environment.

Parameters **game: ple.game.base**

The game the PLE environment manipulates and maintains.

fps: int (default: 30)

The desired frames per second we want to run our game at. Typical settings are 30 and 60 fps.

frame_skip: int (default: 1)

The number of times we skip getting observations while repeat an action.

num_steps: int (default: 1)

The number of times we repeat an action.

reward_values: dict

This contains the rewards we wish to set give our agent based on different actions in game. The current defaults are as follows:

```
rewards = {
    "positive": 1.0,
    "negative": -1.0,
    "tick": 0.0,
    "loss": -5.0,
    "win": 5.0
}
```

Tick is given to the agent at each game step. You can selectively adjust the rewards by passing a dictonary with the key you want to change. Eg. If we want to adjust the negative reward and the tick reward we would pass in the following:

```
rewards = {
    "negative": -2.0,
    "tick": -0.01
}
```

Keep in mind that the tick is applied at each frame. If the game is running at 60fps the agent will get a reward of 60*tick.

force_fps: bool (default: True)

If False PLE delays between game.step() calls to ensure the fps is specified. If not PLE passes an elapsed time delta to ensure the game steps by an amount of time consistent with the specified fps. This is usally set to True as it allows the game to run as fast as possible which speeds up training.

display_screen: bool (default: False)

If we draw updates to the screen. Disabling this speeds up interation speed. This can be toggled to True during testing phases so you can observe the agents progress.

add_noop_action: bool (default: True)

This inserts the NOOP action specified as a valid move the agent can make.

NOOP: pygame.constants (default: K_F15)

The key we want our agent to send that represents a NOOP. This is currently set to F15.

state_preprocessor: python function (default: None)

Python function which takes a dict representing game state and returns a numpy array.

rng: numpy.random.RandomState, int, array_like or None. (default: 24)

Number generator which is used by PLE and the games.

act(*action*)

Perform an action on the game. We lockstep frames with actions. If act is not called the game will not run.

Parameters **action** : int

The index of the action we wish to perform. The index usually corresponds to the index item returned by getActionSet().

Returns int

Returns the reward that the agent has accumlated while performing the action.

game_over()

Returns True if the game has reached a terminal state and False otherwise.

This state is game dependent.

Returns bool

getActionSet()

Gets the actions the game supports. Optionally inserts the NOOP action if PLE has add_noop_action set to True.

Returns list of pygame.constants

The agent can simply select the index of the action to perform.

getFrameNumber()

Gets the current number of frames the agent has seen since PLE was initialized.

Returns int

getGameState()
Gets a non-visual state representation of the game.

This can include items such as player position, velocity, ball location and velocity etc.

Returns dict or None

It returns a dict of game information. This greatly depends on the game in question and must be referenced against each game. If no state is available or supported None will be returned back.

getGameStateDims()
Gets the games non-visual state dimensions.

Returns tuple of int or None

Returns a tuple of the state vectors shape or None if the game does not support it.

getScreenDims()
Gets the games screen dimensions.

Returns tuple of int

Returns a tuple of the following format (screen_width, screen_height).

getScreenGrayscale()
Gets the current game screen in Grayscale format. Converts from RGB using relative lumiance.

Returns numpy uint8 array

Returns a numpy array with the shape (width, height).

getScreenRGB()
Gets the current game screen in RGB format.

Returns numpy uint8 array

Returns a numpy array with the shape (width, height, 3).

init()
Initializes the pygame environement, setup the display, and game clock.

This method should be explicitly called.

lives()
Gets the number of lives the agent has left. Not all games have the concept of lives.

Returns int

reset_game()
Performs a reset of the games to a clean initial state.

saveScreen(*filename*)
Saves the current screen to png file.

Parameters **filename** : string

The path with filename to where we want the image saved.

score()
Gets the score the agent currently has in game.

Returns int

2.2 ple.games.base

class ple.games.base.**Game**(*width*, *height*, *actions={}*)

Game base class

ple.games.base.Game(width, height, actions={})

This Game class sets methods all games require. It should be subclassed when creating new games.

Parameters **width: int**

The width of the game screen.

height: int

The height of the game screen.

actions: dict

Contains possible actions that the game responds too. The dict keys are used by the game, while the values are *pygame.constants* referring the keys.

Possible actions dict:

```
>>> from pygame.constants import K_w, K_s
>>> actions = {
>>>     "up": K_w,
>>>     "down": K_s
>>> }
```

adjustRewards(*rewards*)

Adjusts the rewards the game gives the agent

Parameters **rewards** : dict

A dictonary of reward events to float rewards. Only updates if key matches those specified in the init function.

game_over()

Gets the status of the game, returns True if game has hit a terminal state. False otherwise.

This is game dependent.

Returns bool

getActions()

Gets the actions used within the game.

Returns list of *pygame.constants*

getGameState()

Gets a non-visual state representation of the game.

Returns dict or None

dict if the game supports it and None otherwise.

getScore()

Return the current score of the game.

Returns int

The current reward the agent has received since the last init() or reset() call.

getScreenDims()
Gets the screen dimensions of the game in tuple form.

Returns tuple of int

Returns tuple as follows (width, height).

init()
This is used to initialize the game, such reseting the score, lives, and player position.

This is game dependent.

reset()
Wraps the init() function, can be setup to reset certain poritions of the game only if needed.

setRNG(*rng*)
Sets the rng for games.

step(*dt*)
This method steps the game forward one step in time equal to the dt parameter. The game does not run unless this method is called.

Parameters **dt** : integer

This is the amount of time elapsed since the last frame in milliseconds.