1/16/2023

Abhijith M

Zerone Consulting Pvt Ltd

Reinforcement Learning

Python

# Full code

Git [link](https://github.com/PacktPublishing/Hands-On-Reinforcement-Learning-with-Python.git)

# Installation

* **conda install opencv**
* **pip install gym==0.7.0**
* pip install pyglet==v1.3.2
* conda install swig
* pip install box2d-py

# sample code

import gym  
env = gym.make('CartPole-v0')  
env.reset()  
for \_ in range(1000):  
 env.render()  
 env.step(env.action\_space.sample())

## checking environements

from gym import envs  
print(envs.registry.all())

## car learning

import gym  
env = gym.make('CarRacing-v0')  
env.reset()  
for \_ in range(1000):  
 env.render()  
 env.step(env.action\_space.sample())

## learn to walk

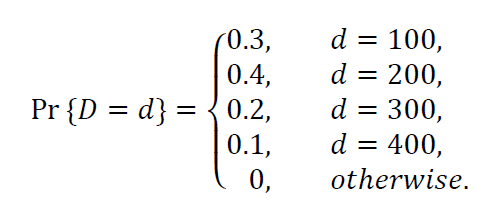
import gym  
env = gym.make('BipedalWalker-v2')  
for i\_episode in range(100):  
 observation = env.reset()  
 for t in range(10000):  
 env.render()  
 print(observation)  
 action = env.action\_space.sample()  
 observation, reward, done, info = env.step(action)  
 if done:  
 print("{} timesteps taken for the episode".format(t+1))  
 break

# Create custom environment

## Example use case – Inventory replenishment of a food truck

Goal: Our use case involves a food truck business that needs to decide how many burger patties to buy every weekday to replenish its inventory

* Our food truck operates downtown during the weekdays
* Every weekday morning, the owner needs to decide on how many burger patties to buy with the following options: 𝐴𝐴 = {0, 100, 200, 300, 400} . The cost of a single patty is 𝑐𝑐 = $4.
* The food truck can store the patties up to a capacity of 𝐶𝐶 = 400 during the weekdays. However, since the truck does not operate over the weekend, and any inventory unsold by Friday evening spoils, if during a weekday, the number of patties purchased and the existing inventory exceeds the capacity, the excess inventory also spoils.
* Burger demand for any weekday is a random variable 𝐷𝐷 with the following probability mass function:



* The net revenue per burger (after the cost of the ingredients other than the patty) is 𝑏𝑏 = $7 .
* Sales in a day is the minimum of the demand and the available inventory, since the truck cannot sell more burgers than the number of patties available.

So, what we have is a multi-step inventory planning problem and our goal is to maximize the total expected profit (𝑏𝑏 − 𝑐𝑐 ) in a week.

### code

import numpy as np  
import gym  
  
  
class FoodTruck(gym.Env):  
 def \_\_init\_\_(self):  
 self.v\_demand = [100, 200, 300, 400]  
 self.p\_demand = [0.3, 0.4, 0.2, 0.1]  
 self.capacity = self.v\_demand[-1]  
 self.days = ['Mon', 'Tue', 'Wed',  
 'Thu', 'Fri', "Weekend"]  
 self.unit\_cost = 4  
 self.net\_revenue = 7  
 # the action is the number of patties to purchase before the sales start.  
 self.action\_space = [0, 100, 200, 300, 400]  
 # The state is a tuple of the day of the week (or the weekend) and the starting inventory level for the day.  
 # Possible inventory levels are 0, 100, 200, and 300  
 # at the beginning of a given day (because of how we defined the action set, possible  
 # demand scenarios, and the capacity); except we start Monday with no inventory.  
 # [('Mon', 0), ('Tue', 0), ('Tue', 100), ('Tue', 200), ('Tue', 300), ('Wed', 0), ('Wed', 100) ... ]  
 self.state\_space = [("Mon", 0)] + [(d, i) for d in self.days[1:] for i in [0, 100, 200, 300]]  
  
 def get\_next\_state\_reward(self, state, action, demand):  
 *"""  
 method that calculates the next state and the reward along with  
 the relevant quantities, given the current state, the action, and the demand. Note  
 that this method does not change anything in the object* ***:param*** *state: state space a tupple of day and possible inventory* ***:param*** *action: number of patties to purchase* ***:param*** *demand: demand on the day* ***:return****: dictionary of values (next\_day, starting\_inventory, cost, sales, revenue, next\_inventory, reward)  
 """* day, inventory = state  
 result = {}  
 result['next\_day'] = self.days[self.days.index(day) + 1]  
 result['starting\_inventory'] = min(self.capacity, inventory + action)  
 result['cost'] = self.unit\_cost \* action  
 result['sales'] = min(result['starting\_inventory'], demand)  
 result['revenue'] = self.net\_revenue \* result['sales']  
 result['next\_inventory'] = result['starting\_inventory'] - result['sales']  
 result['reward'] = result['revenue'] - result['cost']  
 return result  
  
 def get\_transition\_prob(self, state, action):  
 *"""  
 Notice that different demand scenarios will  
 lead to the same next state and reward if the demand exceeds the inventory* ***:param*** *state:* ***:param*** *action:* ***:return****: next\_s\_r\_prob  
 """* next\_s\_r\_prob = {}  
 for ix, demand in enumerate(self.v\_demand):  
 result = self.get\_next\_state\_reward(state, action, demand)  
 next\_s = (result['next\_day'], result['next\_inventory'])  
 reward = result['reward']  
 prob = self.p\_demand[ix]  
 if (next\_s, reward) not in next\_s\_r\_prob:  
 next\_s\_r\_prob[next\_s, reward] = prob  
 else:  
 next\_s\_r\_prob[next\_s, reward] += prob  
 return next\_s\_r\_prob  
  
 def reset(self):  
 *"""  
 Create a reset method, which simply initializes/resets the object to Monday  
 morning with zero inventory. We will call this method before we start an episode,  
 every time:* ***:return****:  
 """* self.day = "Mon"  
 self.inventory = 0  
 state = (self.day, self.inventory)  
 return state  
  
 def is\_terminal(self, state):  
 *"""  
 method to check if a given state is terminal or not. Remember that  
 episodes terminate at the end of the week in this example* ***:param*** *state:* ***:return****:  
 """* day, inventory = state  
 if day == "Weekend":  
 return True  
 else:  
 return False  
  
 def step(self, action):  
 *"""  
 step method that simulates the environment for a one-time step  
 given the current state and the action:  
 The method returns the new state, one-step reward, whether the episode is  
 complete, and any additional information we would like to return. This is the  
 standard Gym convention. It also updates the state stored within the class* ***:param*** *action:* ***:return****:  
 """* demand = np.random.choice(self.v\_demand,  
 p=self.p\_demand)  
 result = self.get\_next\_state\_reward((self.day, self.inventory), action, demand)  
 self.day = result['next\_day']  
 self.inventory = result['next\_inventory']  
 state = (self.day, self.inventory)  
 reward = result['reward']  
 done = self.is\_terminal(state)  
 info = {'demand': demand, 'sales': result['sales']}  
 return state, reward, done, info  
  
  
def base\_policy(states):  
 *"""  
 function that returns a policy dictionary, in which the keys  
 correspond to the states. The value that corresponds to a state is another dictionary  
 that has actions as the keys and the probability of selecting that action in that state  
 as the values:* ***:param*** *states:* ***:return****:  
 """* policy = {}  
 for s in states:  
 day, inventory = s  
 prob\_a = {}  
 if inventory >= 300:  
 prob\_a[0] = 1  
 else:  
 prob\_a[200 - inventory] = 0.5  
 prob\_a[300 - inventory] = 0.5  
 policy[s] = prob\_a  
 return policy  
  
  
def expected\_update(env, v, s, prob\_a, gamma):  
 *"""  
 function that will calculate the expected  
 update for a given state and the corresponding policy for that state:* ***:param*** *env:* ***:param*** *v:* ***:param*** *s:* ***:param*** *prob\_a:* ***:param*** *gamma:* ***:return****:  
 """* expected\_value = 0  
 for a in prob\_a:  
 prob\_next\_s\_r = env.get\_transition\_prob(s, a)  
 for next\_s, r in prob\_next\_s\_r:  
 expected\_value += prob\_a[a] \* prob\_next\_s\_r[next\_s, r] \* (r + gamma \* v[next\_s])  
 return expected\_value  
  
  
def policy\_evaluation(env, policy, max\_iter=100, v=None, eps=0.1, gamma=1):  
 if not v:  
 v = {s: 0 for s in env.state\_space}  
 k = 0  
 while True:  
 max\_delta = 0  
 for s in v:  
 if not env.is\_terminal(s):  
 v\_old = v[s]  
 prob\_a = policy[s]  
 v[s] = expected\_update(env, v, s, prob\_a, gamma)  
 max\_delta = max(max\_delta,  
 abs(v[s] - v\_old))  
 k += 1  
 if max\_delta < eps:  
 print("Converged in", k, "iterations.")  
 break  
 elif k == max\_iter:  
 print("Terminating after", k, "iterations.")  
 break  
 return v  
  
  
def choose\_action(state, policy):  
 prob\_a = policy[state]  
 action = np.random.choice(a=list(prob\_a.keys()),  
 p=list(prob\_a.values()))  
 return action  
  
  
def simulate\_policy(policy, n\_episodes):  
 np.random.seed(0)  
 foodtruck = FoodTruck()  
 rewards = []  
 for i\_episode in range(n\_episodes):  
 state = foodtruck.reset()  
 done = False  
 ep\_reward = 0  
 while not done:  
 action = choose\_action(state, policy)  
 state, reward, done, info = foodtruck.step(action)  
 ep\_reward += reward  
 rewards.append(ep\_reward)  
 print("Expected weekly profit:", np.mean(rewards))  
  
  
def policy\_improvement(env, v, s, actions, gamma):  
 prob\_a = {}  
 if not env.is\_terminal(s):  
 max\_q = np.NINF  
 best\_a = None  
 for a in actions:  
 q\_sa = expected\_update(env, v, s, {a: 1}, gamma)  
 if q\_sa >= max\_q:  
 max\_q = q\_sa  
 best\_a = a  
 prob\_a[best\_a] = 1  
 else:  
 max\_q = 0  
 return prob\_a, max\_q  
  
  
def policy\_iteration(env, eps=0.1, gamma=1):  
 np.random.seed(1)  
 states = env.state\_space  
 actions = env.action\_space  
 policy = {s: {np.random.choice(actions): 1} for s in states}  
 v = {s: 0 for s in states}  
 while True:  
 v = policy\_evaluation(env, policy, v=v, eps=eps, gamma=gamma)  
 old\_policy = policy  
 policy = {}  
 for s in states:  
 policy[s], \_ = policy\_improvement(env, v, s, actions, gamma)  
 if old\_policy == policy:  
 break  
 print("Optimal policy found!")  
 return policy, v  
  
  
foodtruck = FoodTruck()  
policy = base\_policy(foodtruck.state\_space)  
v = policy\_evaluation(foodtruck, policy)  
print("Expected weekly profit:", v["Mon", 0])  
  
simulate\_policy(policy, 1000)  
  
policy, v = policy\_iteration(foodtruck)  
print("Expected weekly profit:", v["Mon", 0])

output

Converged in 6 iterations.

Expected weekly profit: 2515.0

Expected weekly profit: 1389.02

Converged in 6 iterations.

Converged in 6 iterations.

Converged in 5 iterations.

Optimal policy found!

Expected weekly profit: 2880.0

## Implementing first-visit Monte Carlo estimation of the state values

def first\_visit\_return(returns, trajectory, gamma):  
 G = 0  
 T = len(trajectory) - 1  
 for t, sar in enumerate(reversed(trajectory)):  
 s, a, r = sar  
 G = r + gamma \* G  
 first\_visit = True  
 for j in range(T - t):  
 if s == trajectory[j][0]:  
 first\_visit = False  
 if first\_visit:  
 if s in returns:  
 returns[s].append(G)  
 else:  
 returns[s] = [G]  
 return returns  
  
  
def get\_trajectory(env, policy):  
 trajectory = []  
 state = env.reset()  
 done = False  
 sar = [state]  
 while not done:  
 action = choose\_action(state, policy)  
 state, reward, done, info = env.step(action)  
 sar.append(action)  
 sar.append(reward)  
 trajectory.append(sar)  
 sar = [state]  
 return trajectory  
  
  
def first\_visit\_mc(env, policy, gamma, n\_trajectories):  
 np.random.seed(0)  
 returns = {}  
 v = {}  
 for i in range(n\_trajectories):  
 trajectory = get\_trajectory(env, policy)  
 returns = first\_visit\_return(returns,  
 trajectory,  
 gamma)  
 for s in env.state\_space:  
 if s in returns:  
 v[s] = np.round(np.mean(returns[s]), 1)  
 return v  
  
  
foodtruck = FoodTruck()  
policy = base\_policy(foodtruck.state\_space)  
v\_est = first\_visit\_mc(foodtruck, policy, 1, 1000)  
print(v\_est)

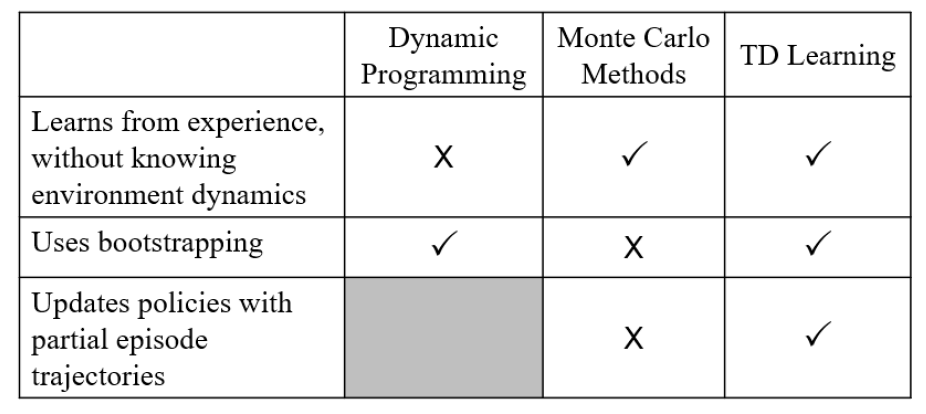
{('Mon', 0): 2518.1, ('Tue', 0): 1966.8, ('Tue', 100): 2374.3, ('Tue', 200): 2802.8, ('Wed', 0): 1441.5, ('Wed', 100): 1824.4, ('Wed', 200): 2187.3, ('Thu', 0): 870.3, ('Thu', 100): 1264.4, ('Thu', 200): 1687.3, ('Fri', 0): 299.6, ('Fri', 100): 725.6, ('Fri', 200): 1151.0}

## On-policy Monte Carlo control

def get\_eps\_greedy(actions, eps, a\_best):  
 prob\_a = {}  
 n\_a = len(actions)  
 for a in actions:  
 if a == a\_best:  
 prob\_a[a] = 1 - eps + eps / n\_a  
 else:  
 prob\_a[a] = eps / n\_a  
  
 return prob\_a  
  
  
def get\_random\_policy(states, actions):  
 policy = {}  
 n\_a = len(actions)  
 for s in states:  
 policy[s] = {a: 1 / n\_a for a in actions}  
 return policy  
  
  
import operator  
  
  
def on\_policy\_first\_visit\_mc(env, n\_iter, eps, gamma):  
 np.random.seed(0)  
 states = env.state\_space  
 actions = env.action\_space  
 policy = get\_random\_policy(states, actions)  
 Q = {s: {a: 0 for a in actions} for s in states}  
 Q\_n = {s: {a: 0 for a in actions} for s in states}  
 for i in range(n\_iter):  
 if i % 10000 == 0:  
 print("Iteration:", i)  
 trajectory = get\_trajectory(env, policy)  
 G = 0  
 T = len(trajectory) - 1  
 for t, sar in enumerate(reversed(trajectory)):  
 s, a, r = sar  
 G = r + gamma \* G  
 first\_visit = True  
 for j in range(T - t):  
 s\_j = trajectory[j][0]  
 a\_j = trajectory[j][1]  
 if (s, a) == (s\_j, a\_j):  
 first\_visit = False  
 if first\_visit:  
 Q[s][a] = Q\_n[s][a] \* Q[s][a] + G  
 Q\_n[s][a] += 1  
 Q[s][a] /= Q\_n[s][a]  
 a\_best = max(Q[s].items(),  
 key=operator.itemgetter(1))[0]  
 policy[s] = get\_eps\_greedy(actions,  
 eps,  
 a\_best)  
 return policy, Q, Q\_n  
  
  
  
foodtruck = FoodTruck()  
policy, Q, Q\_n = on\_policy\_first\_visit\_mc(foodtruck,  
 300000,  
 0.05,  
 1)

## off-policy Monte Carlo

def off\_policy\_mc(env, n\_iter, eps, gamma):  
 np.random.seed(0)  
 states = env.state\_space  
 actions = env.action\_space  
 Q = {s: {a: 0 for a in actions} for s in states}  
 C = {s: {a: 0 for a in actions} for s in states}  
 target\_policy = {}  
 behavior\_policy = get\_random\_policy(states,  
 actions)  
 for i in range(n\_iter):  
 if i % 10000 == 0:  
 print("Iteration:", i)  
 trajectory = get\_trajectory(env, behavior\_policy)  
 G = 0  
 W = 1  
 T = len(trajectory) - 1  
 for t, sar in enumerate(reversed(trajectory)):  
 s, a, r = sar  
 G = r + gamma \* G  
 C[s][a] += W  
 Q[s][a] += (W / C[s][a]) \* (G - Q[s][a])  
 a\_best = max(Q[s].items(), key=operator.itemgetter(1))[0]  
 target\_policy[s] = a\_best  
 behavior\_policy[s] = get\_eps\_greedy(actions,  
 eps,  
 a\_best)  
 if a != target\_policy[s]:  
 break  
 W = W / behavior\_policy[s][a]  
 target\_policy = {s: target\_policy[s] for s in states}  
 return target\_policy, Q  
  
  
foodtruck = FoodTruck()  
policy, Q = off\_policy\_mc(foodtruck, 300000, 0.05, 1)  
print(policy)



## One-step TD learning – TD(0)

def one\_step\_td\_prediction(env, policy, gamma, alpha, n\_iter):  
 np.random.seed(0)  
 states = env.state\_space  
 v = {s: 0 for s in states}  
 s = env.reset()  
 for i in range(n\_iter):  
 a = choose\_action(s, policy)  
 s\_next, reward, done, info = env.step(a)  
 v[s] += alpha \* (reward + gamma \* v[s\_next] -  
 v[s])  
 if done:  
 s = env.reset()  
 else:  
 s = s\_next  
 return v  
  
  
foodtruck = FoodTruck()  
policy = base\_policy(foodtruck.state\_space)  
out = one\_step\_td\_prediction(foodtruck, policy, 1, 0.01, 100000)  
print(out)

{('Mon', 0): 2506.576417395407, ('Tue', 0): 1956.077876400167, ('Tue', 100): 2368.7400039407535, ('Tue', 200): 2767.5069659225423, ('Tue', 300): 0, ('Wed', 0): 1413.0055559001296, ('Wed', 100): 1813.546186490315, ('Wed', 200): 2200.8873259700867, ('Wed', 300): 0, ('Thu', 0): 828.2915189850011, ('Thu', 100): 1280.424626614422, ('Thu', 200): 1675.8661846955831, ('Thu', 300): 0, ('Fri', 0): 345.52991944823583, ('Fri', 100): 677.4358179389413, ('Fri', 200): 1094.8263154150825, ('Fri', 300): 0, ('Weekend', 0): 0, ('Weekend', 100): 0, ('Weekend', 200): 0, ('Weekend', 300): 0}

## On-policy control with SARSA

def sarsa(env, gamma, eps, alpha, n\_iter):  
 np.random.seed(0)  
 states = env.state\_space  
 actions = env.action\_space  
 Q = {s: {a: 0 for a in actions} for s in states}  
 policy = get\_random\_policy(states, actions)  
 s = env.reset()  
 a = choose\_action(s, policy)  
 for i in range(n\_iter):  
 if i % 100000 == 0:  
 print("Iteration:", i)  
 s\_next, reward, done, info = env.step(a)  
 a\_best = max(Q[s\_next].items(),  
 key=operator.itemgetter(1))[0]  
 policy[s\_next] = get\_eps\_greedy(actions, eps, a\_best)  
 a\_next = choose\_action(s\_next, policy)  
 Q[s][a] += alpha \* (reward  
 + gamma \* Q[s\_next][a\_next]  
 - Q[s][a])  
 if done:  
 s = env.reset()  
 a\_best = max(Q[s].items(),  
 key=operator.itemgetter(1))[0]  
 policy[s] = get\_eps\_greedy(actions, eps, a\_best)  
 a = choose\_action(s, policy)  
 else:  
 s = s\_next  
 a = a\_next  
  
 return policy, Q  
  
  
foodtruck = FoodTruck()  
policy, Q = sarsa(foodtruck, 1, 0.1, 0.05, 1000000)  
print(policy)

{('Mon', 0): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Tue', 0): {0: 0.92, 100: 0.02, 200: 0.02, 300: 0.02, 400: 0.02}, ('Tue', 100): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Tue', 200): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Tue', 300): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Wed', 0): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Wed', 100): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Wed', 200): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Wed', 300): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Thu', 0): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Thu', 100): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Thu', 200): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Thu', 300): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Fri', 0): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Fri', 100): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Fri', 200): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Fri', 300): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Weekend', 0): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Weekend', 100): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Weekend', 200): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}, ('Weekend', 300): {0: 0.2, 100: 0.2, 200: 0.2, 300: 0.2, 400: 0.2}}

## Off-policy control with Q-learning

def q\_learning(env, gamma, eps, alpha, n\_iter):  
 np.random.seed(0)  
 states = env.state\_space  
 actions = env.action\_space  
 Q = {s: {a: 0 for a in actions} for s in states}  
 policy = get\_random\_policy(states, actions)  
 s = env.reset()  
  
 for i in range(n\_iter):  
 if i % 100000 == 0:  
 print("Iteration:", i)  
 a\_best = max(Q[s].items(),  
 key=operator.itemgetter(1))[0]  
 policy[s] = get\_eps\_greedy(actions, eps, a\_best)  
 a = choose\_action(s, policy)  
 s\_next, reward, done, info = env.step(a)  
 Q[s][a] += alpha \* (reward  
 + gamma \* max(Q[s\_next].  
 values())  
 - Q[s][a])  
 if done:  
 s = env.reset()  
 else:  
 s = s\_next  
 policy = {s: {max(policy[s].items(),  
 key=operator.itemgetter(1))[0]: 1}  
 for s in states}  
 return policy, Q  
  
  
foodtruck = FoodTruck()  
policy, Q = q\_learning(foodtruck, 1, 0.1, 0.01, 1000000)

# Deep Reinforcement Learning

## Documentations

[link](https://www.youtube.com/watch?v=doR5bMe-Wic&t=3957s)

comparison of Ray to other distributed backend [frameworks](https://bit.ly/2T44AzK)

Ray [documentation](https://docs.ray.io/en/latest/index.html) source [code](https://github.com/ray-project/ray)

# Ray

## Installation

* pip install -U ray

## Ray implementation of a DQN variant

• train\_apex\_dqn.py is the main script that accepts the training configs and

initializes the other components.

• actor.py includes the RL actor class that interacts with the environment and

collects experiences.

• parameter\_server.py includes a parameter server class that serves the

optimized Q model weights to actors.

• replay.py includes the replay buffer class.

• learner.py includes a learner class that receives samples from the replay buffer,

takes gradient steps, and pushes the new Q-network weights to the parameter

server.

• models.py includes functions to create a feedforward neural network using

TensorFlow/Keras.