Functional programming paradigms in Reinforcement learning problems

By: Pietro Ehrlich

July 2022

Abstract

Machine learning, and more specifically, Reinforcement learning, has been one of the areas of computer science with the most promise and has advanced at an accelerated rate since its inception. However, these advancements have come at the cost of sacrificing best practices, especially in the libraries that compromise standards to gear them towards practical use.

One such fact can be noticed in the use of Object-Oriented Programming in the development of Machine learning algorithms since stateful programs tend to be harder to test and grow efficiently and have an ever-growing amount of side effects in every process. This is why this thesis attempts to create a Reinforcement learning library that is purely functional using Racket.

Chapter 1

Introduction

With the growing amount of applications of reinforcement learning, the complexity of the projects and, as such, the need for proper software development practices in all the stack of the program is needed more than ever.

In this sense, maintainability of the code and increased developer productivity are prime requirements of teams. These are precisely some of the advantages brought by functional programming. This paradigm is highly used in academia and projects closely related to mathematics because it allows easier demonstration of the correctness of the code and is very useful for lambda calculus.

Due to the advantages of functional programming, this thesis attempts to build a reinforcement learning library built with Racket that makes all the development of the algorithms completely functional. Racket is a multi-paradigm programming language that, however, greatly encourages functional programming since its expression-based structure and its roots as a research language have made it very appropriate to the functional paradigm by design.

Approach

Two classic Reinforcement learning problems were chosen to test the effectiveness of the library Mountain Car and Taxi . These two problems were solved using pure Racket and

then with the Reinforcement learning library to understand the usefulness and impact on the development of the solutions. The goal was to solve reinforcement learning problems using functional programming.

Chapter 2: Background

In this chapter, we are going to introduce the main subjects that are needed to understand the work presented. First, we present what reinforcement learning is and specifically we explain Q-learning, which is the reinforcement learning method used. Next we will look at functional programming and the differences that it has to object oriented programming and Racket and the reasons for which it was chosen.

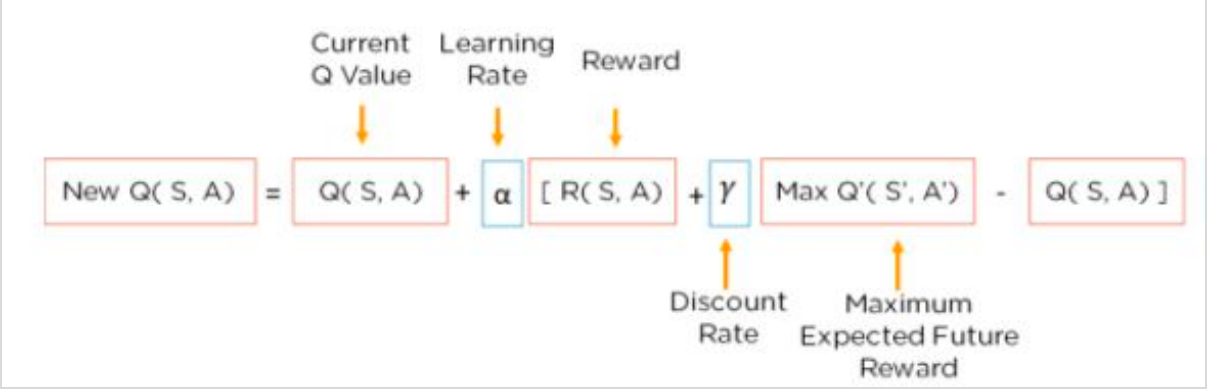
Reinforcement Learning

Reinforcement learning is one of the main areas of study in machine learning, with it being used for autonomous driving, computer vision and many other applications. The basic idea of reinforcement learning is to train an autonomous agent to act autonomously in an environment to maximize a reward. The agent is given a reward after performing each action in an unknown environment and the agent uses the reward to tune its future decisions.

Many algorithms such as SARSA or Q Learning exist to solve problems via reinforcement learning. The technique applied in this thesis is Q Learning.

Q Learning

Q learning seeks to find the best policy by exploration-exploitation. It relies on the bellman equation which is used to determine how good the current state is and by moving between states it improves the valuation of each state.



This equation fills up the values in a Q table that holds the values associated with every state-action pair and is used to decide the best policy.

At each step, the algorithm decides between choosing a random move (exploration) or choosing the best move according to the current policy (exploitation). This decision is determined by an epsilon that is passed to the algorithm and is the probability that exploration will be chosen over exploitation. Since the Q Table is initially empty, we tend to prefer exploration over exploitation so a common optimization technique of Q Learning is to decay the epsilon value over time so that initially a random action is very likely to be chosen and as the policy improves, it follows it more.

Functional programming

Functional programming is a software construction methodology that uses only pure functions, avoiding shared state, mutable data and unknown code dependencies.
The basis of functional programming is that all code is written inside a function and the functions are "pure" which means that all the inputs are declared as inputs and all the outputs are declared as outputs so it does not have any outside dependencies and the code can be tested in isolation.
Object-oriented programming relies instead on a mutable shared state and has an imperative programming model such that statements change the program's state.

Functional programming is commonly used in research since having no outside dependencies in the code allows users to easily prove that it is correct. This also makes functional programming better for debugging since unit tests and finding errors in the code are easier to detect as the function is known and it is also known that the problem is inside the function since it has no outside dependencies.

The modularity of functional programming also means that developers can be more productive since the contract of every function is explicit in its input and output so developers don't need to worry about unknown dependencies in their code changes. This modularity is also very useful in reinforcement learning applications since the application can be easily tested independently by each use.

Racket

Racket is a general-purpose programming language built on top of Lisp and although it is a multi-paradigm language, it highly incentivises functional programming. The language is highly used in research due to its robust set of libraries in fields such as math & statistics

and to create DSLs (domain-specific languages).

Racket was chosen because of its closeness to functional programming and its ability to be used to build languages since, in a further iteration of this work, a DSL to solve reinforcement learning problems could be built. It is also close to the target audience of this work since it is a research project that the academic community that uses Racket could prefer.

Chapter 3: Preparation

Before we build the library we solved two Reinforcement learning problems using pure Racket and without forcing our code to be functional since generally reinforcement learning problems are solved using imperative programming with shared state for variable such as the Q table and so the difference in the functional approach can be seen in a clearer manner when comparing both approaches.

Mountain Car

Mountain car is a classic reinforcement learning problem in which the agent has to learn to drive a car up a two-dimensional sinusoidal valley. The hill is high enough so that the car has to get momentum to reach the end. The car can either move left, move right or not accelerate on each step and has to reach the goal at the end of the valley in the least amount of steps.

Environment

The environment has already been developed by Open AI for Python solutions but no Racket environment exists so we built the environment ourselves. We followed the exact environment specifications built by open AI, namely:

- The action space can be 0 (accelerate to the left), 1 (don't accelerate) and 2 (accelerate to the right)
- The observation space has two dimensions: Position and velocity, the position can go from -1.2 to 0.6 and the velocity is limited between -0.7 and 0.7.
- The original starting state is a random position between -0.6 and -0.4 and the velocity is 0.
- The goal is set at position 0.5.
- The car gets a -1 reward every step since we want it to reach the goal as quickly as possible.
- The maximum length of an episode (a series of steps) is 200.

The code for the environment results in the following:

```
#lang racket

(provide act done reward restart observation action_space observation_space
get_steps)

(define state_position (box (-(* (random) 0.2) 0.6)))
(define state_velocity (box 0.0))
(define step (box 0))

(define (act_velocity action)
  (define new_vel (- (+ (unbox state_velocity) (* (+ action -1) 0.001)) ( *
(cos (* 3 (unbox state_position))) 0.0025)))
  (set! new_vel (max new_vel -0.07))
  (set! new_vel (min new_vel 0.07))
  new_vel
  )

(define (act_position action)
  (define new_pos (+ (unbox state_position) (unbox state_velocity)))
  (when (<= new_pos -1.2)
    (set! new_pos -1.2)
    (set-box! state_velocity 0.0)
    )
  (when (>= new_pos 0.6)
    (set! new_pos 0.6)
    (set-box! state_velocity 0.0)
    )
  new_pos
  )

(define (act action)
  (set-box! state_velocity (act_velocity action))
  (set-box! state_position (act_position action))
  (set-box! step (+ (unbox step) 1))
  )

(define (observation)
```

```
  (cons (unbox state_position) (unbox state_velocity))
  )

(define (done) (or (>= (unbox state_position) 0.5) (>= (unbox step) 200)))

(define (reward) (if (>= (unbox state_position) 0.5) 0 -1))

(define (get_steps) (unbox step))

(define (restart)
  (set-box! state_velocity 0.0)
  (set-box! state_position (- (* (random) 0.2) 0.6))
  (set-box! step 0)
  )

(define (action_space) (list 0 1 2))

(define (observation_space) (cons (cons -1.2 0.6) (cons -0.07 0.07)))
```

We have 4 main parts of the code:

- **The state**: We are keeping three states, the position, the velocity and the steps. The first two allows us to track the agent and the last one is used to understand the episode is done.
- **The action:** It updates the state of the agent according to the given action: we built act_velocity, act_position and act to fulfill this requirement following the original open AI specifications to calculate the new position and velocity.
- **The observation functions:** The model needs to get feedback from the environment after each step, namely, the current state (position and velocity) the reward after the step and if the episode is done
- **Restart:** We need to be able to restart the state after each episode.

Model

The model to solve the problem is a classic Q Learning algorithm with one important change. Since the position and velocity are continuous spaces, we need to discretize the space to be able to create our Q Table that needs to have defined dimensions. We do this by creating equally spaced "buckets" that group the continuous space of that bucket we can map the continuous space to a discrete index of the Q Table. We divide the position in

10 buckets and the velocity in 100 buckets.

In Racket, the code to discretize the space looks like this:

```
(define (discretize_state state space)
  (define bucket_size (cons (/ (- (cdr (car space)) (car (car space))) 10) (/
(- (cdr (cdr space)) (car (cdr space))) 100)))
  (define normalized_state (cons (- (car state) (car (car space))) (- (cdr
state) (car (cdr space)))))

  (cons (inexact->exact(floor (/ (car normalized_state) (car bucket_size))))
(inexact->exact(floor (/ (cdr normalized_state) (cdr bucket_size)))))
)
```

Now we can build our complete Q Learning model:

```
#lang racket

(require "Mountain_Car_Environment.rkt")

(define learning_rate 0.2)
(define discount 0.9)
(define buckets 30)
(define episodes 5000)
(define const_epsilon 0.8)
(define epsilon_end_decay 0)
(define epsilon_decay_value 0.00016)

(define Q (build-vector 11
                        (lambda (i)
                          (build-vector 101 (lambda (j)
                                              (make-vector 3 0))))); position
velocity action

(define (get-Q position velocity)
  (vector-ref (vector-ref Q position) velocity)
  )
```

```
(define (set-Q position velocity action value)
  (vector-set! (get-Q position velocity) action value)
  )

(define (discretize_state state space)
  (define bucket_size (cons (/ (- (cdr (car space)) (car (car space))) 10) (/
(- (cdr (cdr space)) (car (cdr space))) 100)))
  (define normalized_state (cons (- (car state) (car (car space))) (- (cdr
state) (car (cdr space)))))

  (cons (inexact->exact(floor (/ (car normalized_state) (car bucket_size))))
(inexact->exact(floor (/ (cdr normalized_state) (cdr bucket_size)))))
  )

(define (choose_action state epsilon)
  (cond [( >= (random) epsilon)
         (define max_Q (vector-argmax max (get-Q (car state) (cdr state))))
         (vector-member max_Q (get-Q (car state) (cdr state)))
         ]
        [else
         (random 3)
         ])
  )

(define (take_action epsilon)
  (define state (discretize_state (observation) (observation_space)))

  (define action (choose_action state epsilon))

  (act action)
  (define new_state (discretize_state (observation) (observation_space)))

  (cond [(and (done) (= (reward) 0))
         (set-Q (car state) (cdr state) action 0)
         ]
        [else
         (define max_Q (vector-ref (get-Q (car state) (cdr state)) action))
         (define max_future_Q (vector-argmax max (get-Q (car new_state) (cdr
new_state))))
```

```
          (set-Q (car state) (cdr state) action (+ (* (- 1 learning_rate) max_Q)
(* (+ (* discount max_future_Q) (reward)) learning_rate)))
          ]
         )

  (if (done)
      (display (~a "Number of steps: " (get_steps) "\n"))
      (play epsilon))
  )

(define (play epsilon)
  (if (not (done))
      (take_action epsilon)
      (restart)
      )
  )

(define (run_episodes episodes epsilon end_decay decay_value)
  (define new_epsilon epsilon)

  (when (>= episodes end_decay)
    (set! new_epsilon (- epsilon decay_value))
    )

  (unless (zero? episodes)
    (play new_epsilon)
    (run_episodes (- episodes 1 ) new_epsilon end_decay decay_value)
    )
  )

(run_episodes episodes const_epsilon epsilon_end_decay epsilon_decay_value)
```

Our first step is to define our constants and variables for the model, from the learning rate to the Q table which is a 3 dimensional vector filled with zeros. We then have a couple utility functions to access and update our Q vector to simplify our code. The rest of the code is the actual Q learning, we have `choose_action` to take the appropriate action according to the epsilon, `take_action` is the function that takes each step and updates the Q table according to the response from the environment. `play` runs a full episode by

recurrently calling take_action or restarting the environment and `run_episodes` gets recursively called to run the given number of episodes we need to train it.

Taxi

Taxi is the second problem that we are going to tackle to test our functional programming library is the taxi problem which is also part of the Open AI gym. The goal is to move the taxi to the passenger, pick him up, and drop him at the dropout coordinates.

Environment

The taxi moves in a grid and starts in a random square. The passenger's location and the dropoff location are a random position in one of 4 possible coordinates. The grid is always the same and has walls that cannot be crossed.
The taxi actions are moving in any of the four directions, picking up a passenger or dropping him off.
The agent is rewarded -1 per step unless he successfully delivers a passenger which gives a 20 reward or he illegally executes a dropoff or pickup action which gives rewards of -10.

The environment then looks like so:

```
#lang racket

(provide take_action observation restart)

(define matrix (vector-immutable
                (vector-immutable "" ":" "" "|" "" ":" "" ":" "")
                (vector-immutable "" ":" "" "|" "" ":" "" ":" "")
                (vector-immutable "" ":" "" ":" "" ":" "" ":" "")
                (vector-immutable "" "|" "" ":" "" "|" "" ":" "")
                (vector-immutable "" "|" "" ":" "" "|" "" ":" "")
                ))

(define origin_locations (vector-immutable (cons 0 0) (cons 0 4) (cons 4 0)
(cons 4 3)))
(define passenger_position (box (vector-ref origin_locations (random 4))))
(define dropoff_position (box (vector-ref origin_locations (random 4))))

(define taxi_position (box (cons (random 5) (random 5))))
```

```
(define has_passenger (box #f))

(define (validate_move row column)
  (and (>= row 0) (< row 5) (>= column 0) (< column 5))
  )

(define (move_has_no_wall row column new_col)
  (define vec (vector-ref matrix row))
  (string=? ":" (vector-ref vec (- (* (max column new_col) 2) 1)))
  )

(define (equal-pairs a b)
  (and (= (car a) (car b)) (= (cdr a) (cdr b)))
  )

(define (act action)
  (define row (car (unbox taxi_position)))
  (define column (cdr (unbox taxi_position)))

  (cond
    [(= action 0)
     (if (validate_move (+ row 1) column)
         (set-box! taxi_position (cons (+ row 1) column))
         -1
         )
     -1
     ]
    [(= action 1)
     (if (validate_move (- row 1) column)
         (set-box! taxi_position (cons (- row 1) column))
         -1
         )
     -1
     ]
    [(= action 2)
     (if (and (validate_move row (+ column 1)) (move_has_no_wall row column (+
column 1)))
         (set-box! taxi_position (cons row (+ column 1)))
         -1
```

```
          )
      -1
      ]
     [(= action 3)
      (if (and (validate_move row (- column 1)) (move_has_no_wall row column (-
column 1)))
          (set-box! taxi_position (cons row (- column 1)))
          -1
          )
      -1
      ]
     [(= action 4)
      (if (and (equal-pairs (unbox taxi_position) (unbox passenger_position))
(not (unbox has_passenger)))
          (begin
            (set-box! has_passenger #t)
            -1)
          -10
          )
      ]
     [(= action 5)
      (if (and (equal-pairs (unbox taxi_position) (unbox dropoff_position))
(unbox has_passenger))
          (begin
            (set-box! has_passenger #f)
            20)
          -10
          )
      ]
     )
  )

(define (take_action action)
  (act action)
  )

(define (get_position_index position is_passenger)
  (cond
    [(and (unbox has_passenger) is_passenger)
```

```
      4
      ]
     [(equal-pairs position (vector-ref origin_locations 0))
      0
      ]
     [(equal-pairs position (vector-ref origin_locations 1))
      1
      ]
     [(equal-pairs position (vector-ref origin_locations 2))
      2
      ]
     [(equal-pairs position (vector-ref origin_locations 3))
      3
      ]
     )
  )

(define (observation) (list (unbox taxi_position) (get_position_index (unbox
passenger_position) #t) (get_position_index (unbox dropoff_position) #f)))

(define (restart)
  (set-box! taxi_position (cons (random 5) (random 5)) )
  (set-box! has_passenger #f)
  )
```

It has the same 4 main parts of the code:

- **The state**: The state includes the grid, the taxi position, the pickup and drop-off locations.
- **The action:** It updates the state of the agent according to the given action it has to be a valid move that does not cross a wall and stays between the bounds of the grid and also returns the appropriate reward.
- **The observation functions:** The model needs to get feedback from the environment after each step, namely, the current state (taxi position, passenger position and drop-off position.
- **Restart:** We need to be able to restart the state after each episode.

Model

In this Q Learning algorithm it can be applied directly without any changes. We build a Q

Table with 5 dimensions: the taxi row, taxi column, the passenger location, the drop-off location and the action.

The rest of the model is a standard Q Learning algorithm which we can see here:

```
#lang racket

(require "Taxi_Environment.rkt")

(define learning_rate 0.1)
(define discount 0.9)
(define episodes 20000)
(define epsilon 1)
(define end_decay 10000)
(define decay_rate 0.000095)
(define max_steps 40)

(define current_step (box 0))
(define total_reward (box 0))
(define all_rewards (box (make-vector episodes 100)))

(define (same c) c)

(define Q (build-vector 5
                        (lambda (i)
                          (build-vector 5 (lambda (j)
                                            (build-vector 5 (lambda (k)
                                                              (build-vector 4
(lambda (l)

(make-vector 6 0)))
                                                              ))))))); (taxi
row; taxi column; passenger location; dropoff location; action)

(define (get-Q taxi_row taxi_column passenger_location dropoff_location)
  (vector-ref (vector-ref (vector-ref (vector-ref Q taxi_row) taxi_column)
passenger_location) dropoff_location)
  )
```

```
(define (set-Q taxi_row taxi_column passenger_location dropoff_location action
value)
  (vector-set! (get-Q taxi_row taxi_column passenger_location dropoff_location)
action value)
  )

(define (choose_action state epsilon)
  (cond [( >= (random) epsilon)
         (define Q_state (get-Q (car (list-ref state 0)) (cdr (list-ref state
0)) (list-ref state 1) (list-ref state 2)))
         (define max_Q (vector-argmax same Q_state))
         (vector-member max_Q Q_state)
         ]
        [else
         (random 6)
         ])
  )

(define (save_end_of_episode)
  (display (~a "Number of steps: " (unbox current_step) "\n" "Reward: " (unbox
total_reward) "\n"))
  (define index (vector-member 100 (unbox all_rewards)))
  (vector-set! (unbox all_rewards) index (unbox total_reward))
  )

(define (play_episode epsilon)
  (define state (observation))

  (define action (choose_action state epsilon))

  (define reward (take_action action))
  (define new_state (observation))

  (define Q_state (get-Q (car (list-ref state 0)) (cdr (list-ref state 0))
(list-ref state 1) (list-ref state 2)))

  (define max_Q (vector-ref Q_state action))
  (define max_future_Q (vector-argmax same Q_state))
  (set-Q (car (list-ref state 0)) (cdr (list-ref state 0)) (list-ref state 1)
```

```
(list-ref state 2) action (+ (* (- 1 learning_rate) max_Q) (* (+ (* discount
max_future_Q) reward) learning_rate)))

  (set-box! total_reward (+ (unbox total_reward) reward))
  (set-box! current_step (+ (unbox current_step) 1))

  (if (or (= reward 20) (>= (unbox current_step) max_steps))
      (save_end_of_episode)
      (play epsilon reward))
  )

(define (play epsilon reward)
  (when (and (< reward 20) (<= (unbox current_step) max_steps))
    (play_episode epsilon)
    )
  )

(define (run_episodes episodes epsilon end_decay decay_value)
  (define new_epsilon epsilon)

  (when (>= episodes end_decay)
    (set! new_epsilon (- epsilon decay_value))
    )

  (set-box! total_reward 0)
  (set-box! current_step 0)
  (restart)

  (unless (zero? episodes)
    (play new_epsilon 0)
    (run_episodes (- episodes 1 ) new_epsilon end_decay decay_value)
    )
  )

(run_episodes episodes epsilon end_decay decay_rate)
```

Note that we are running very similar functions as in the Mountain Car problem such as the choice of action, the approach to repeating episodes and steps and the process to take a

step. Most of the differences are actually defined in the environment. We are going to try to abstract these similarities into our own library.

Chapter 4: Implementation

As we saw, Q Learning algorithms generally have an imperative programming paradigm in which the Q table and other variables are declared as global states. However, we also know the advantages of functional programming in testing the correctness of code and the modularity which allows for more productivity and clearer code.

Because the library will be completely functional, we will need to define as inputs all the variations of the library. This includes the common variables such as epsilon, discount, learning rate, q, number of episodes, number of steps and epsilon decay. However, we also will need to define as inputs two extra parameters:

- `Q_handler`: As we saw in Mountain Car, the observation or action space can be continuous and as such, accessing the Q table might not be trivial and it will depend on the environment. The handler will be a function that will return the q value at a given state and action or update it if an optional parameter for a new value is given.
  Because of this, we must pass this parameter that will be a function with the following structure:

```
(Q_handler Q state action [new_value (void)])
Q = the Q table that was passed with the Q handler (it has to be passed each
time since pure functions have no "memory", each call is the same as the first
time it was called so we need to give it the handler.
state = the state that will be used as indexes of the table.
action = the action that will be used as indexes of the table.
new_value = an optional parameter that indicates that the table should be
updated with the given value.
```

- `environment`: This is the environment that responds to the users actions, it is given a state and action and returns the new state, the reward and if the episode is finished.

The implementation of Q learning Is based on recursively running each step and iterating through all of the episodes while accumulating the reward and the Q table values to return as the result. The implementation is the following:

```
#lang racket

(define (get_action_space Q)
  (vector-length (vector-ref (vector-ref Q 0) 0))
  )

(define (choose_action state Q Q_handler epsilon)
  (define action_space (get_action_space Q))
  (cond [( >= (random) epsilon)
         (define action_Qs (for/list ([i (in-range action_space)]) (Q_handler Q
state i)))
         (define max_Q (argmax max action_Qs))
         (index-of action_Qs max_Q)
         ]
        [else
         (random action_space)
         ])
  )

(define (take_step state environment Q Q_handler epsilon learning_rate
discount)
  (define action (choose_action state Q Q_handler epsilon))
  (define env_result (environment state action))

  (define curr_Q (Q_handler Q state action))
  (define max_future_Q (argmax max (for/list ([i (in-range (get_action_space
Q))]) (Q_handler Q state i))))
  (define new_Q (Q_handler Q state action (+ (* (- 1 learning_rate) curr_Q) (*
(+ (* discount max_future_Q) (second env_result)) learning_rate))))
  (list (second env_result) new_Q (first env_result) (third env_result)) ;
Reward Q State Done
  )

(define (run_episode steps state environment Q Q_handler epsilon learning_rate
discount)
  (define episodes_results (list Q 0))
```

```
  (define step_result (take_step state environment Q Q_handler epsilon
learning_rate discount))

  (if (or (= steps 1) (fourth step_result))
      (list (second step_result) (first step_result))
      (begin
        (set! episodes_results (run_episode (- steps 1) (third step_result)
environment (second step_result) Q_handler epsilon learning_rate discount))
        (list (first episodes_results) (+ (first step_result) (second
episodes_results)))
        )
      ) ; Q reward
  )

(define (q_learning episodes max_steps initial_state environment Q Q_handler
epsilon epsilon_decay_value learning_rate discount)
  (define all_rewards empty)
  (define new_Q Q)
  (define new_epsilon epsilon)
  (define episode_result (list Q all_rewards))


  (for/list ([i (in-range episodes)])
    (begin
      (when (> epsilon_decay_value 0)
        (set! new_epsilon (max (- new_epsilon epsilon_decay_value) 0))
        )
      (set! episode_result (run_episode max_steps initial_state environment
new_Q Q_handler new_epsilon learning_rate discount))
      (set! new_Q (first episode_result))
      (set! all_rewards (append all_rewards (list (second episode_result))))
      )
    )

  (list Q all_rewards)
  )
```

With the only public function being `q_learning` which is the one that is called to solve

problems.

Mountain Car

To solve any problem the main steps are creating the environment function which corresponds closely to the defined take_action functions and we also have to build the Q table and the Q handler which in this case has the job of discretizing the state to access the Q Table. The implementation is as simple as the following:

```
(define (act_velocity state action)
  (define new_vel (+ (cdr state) (- (* (+ action -1) 0.001) ( * (cos (* 3 (car
state))) 0.0025))))
  (set! new_vel (max new_vel -0.07))
  (set! new_vel (min new_vel 0.07))
  (cons (car state) new_vel)
  )

(define (act_position state action)
  (define new_pos (+ (car state) (cdr state)))
  (define new_vel (cdr state))
  (when (<= new_pos -1.2)
    (set! new_pos -1.2)
    (set! new_vel 0.0)
    )
  (cons new_pos new_vel)
  )

(define (mountain_car_environment state action)
  (define new_state state)
  (set! new_state (act_velocity new_state action))
  (set! new_state (act_position new_state action))
  (list new_state -1 (>= (car new_state) 0.5)) ; State Reward Done
  )

(define (discretize_state state)
  (define space (cons (cons -1.2 0.6) (cons -0.07 0.07)))
  (define bucket_size (cons (/ (- (cdr (car space)) (car (car space))) 10) (/
(- (cdr (cdr space)) (car (cdr space))) 100)))
  (define normalized_state (cons (- (car state) (car (car space))) (- (cdr
state) (car (cdr space)))))
```

```
  (cons (inexact->exact(round (/ (car normalized_state) (car bucket_size))))
(inexact->exact(round (/ (cdr normalized_state) (cdr bucket_size)))))
  )

(define (Q_handler Q state action [new_value (void)])
  (define disc_state (discretize_state state))
  (define action_Q (vector-ref (vector-ref Q (car disc_state)) (cdr
disc_state)))
  (if (void? new_value)
      (vector-ref action_Q action)
      (begin
        (vector-set! action_Q action new_value)
        Q
        )
      )
  )

(define (run_Q_learning)
  (define episodes 10000)
  (define epsilon 0.8)
  (define epsilon_decay_value 0.00008)
  (define learning_rate 0.2)
  (define discount 0.9)

  (define Q (build-vector 11
                          (lambda (i)
                            (build-vector 101 (lambda (j)
                                                (make-vector 3 0))))))
  (define ans (q_learning episodes 200 (cons -0.5 0) mountain_car_environment Q
Q_handler epsilon epsilon_decay_value learning_rate discount))
  ans
  )

(run_Q_learning)
```

We can see that this implementation is drastically shorter and clearer than I pure racket. Going from both the file of the environment and the file of the model to this three pure

functions. We can see that there is no shared state and all functions have clear input and output without any side effect while also having a complete abstraction of the reinforcement learning implementation but still having flexibility thanks to the environment, q handler and constants that are passed to the q_learning function.
The implementation difference of this problem, which is the discretization of the state is easily managed by the Q handler.

Taxi

The taxi problem has a more complex environment but an easier handling of the state so we can see that the q_handler is almost a direct access of the Q vector being a very simple problem to implement with the library.

The result is the following:

```
(define (validate_move row column)
  (and (>= row 0) (< row 5) (>= column 0) (< column 5))
  )

(define (move_has_no_wall row column new_col)
  (define matrix (vector-immutable
                  (vector-immutable "" ":" "" "|" "" ":" "" ":" "")
                  (vector-immutable "" ":" "" "|" "" ":" "" ":" "")
                  (vector-immutable "" ":" "" ":" "" ":" "" ":" "")
                  (vector-immutable "" "|" "" ":" "" "|" "" ":" "")
                  (vector-immutable "" "|" "" ":" "" "|" "" ":" "")
                  ))
  (define vec (vector-ref matrix row))
  (string=? ":" (vector-ref vec (- (* (max column new_col) 2) 1)))
  )

(define (equal-pairs a b)
  (and (= (car a) (car b)) (= (cdr a) (cdr b)))
  )

(define (taxi_environment state action)
  (define row (car (first state)))
  (define column (cdr (first state)))
  (define origin_locations (vector-immutable (cons 0 0) (cons 0 4) (cons 4 0)
(cons 4 3)))
```

```
  (define ans (cond
                [(= action 0)
                 (if (validate_move (+ row 1) column)
                     (list (cons (+ row 1) column) (second state) (third state)
-1)
                     (list (first state) (second state) (third state) -1)
                     )
                 ]
                [(= action 1)
                 (if (validate_move (- row 1) column)
                     (list (cons (- row 1) column) (second state) (third state)
-1)
                     (list (first state) (second state) (third state) -1)
                     )
                 ]
                [(= action 2)
                 (if (and (validate_move row (+ column 1)) (move_has_no_wall
row column (+ column 1)))
                     (list (cons row (+ column 1)) (second state) (third state)
-1)
                     (list (first state) (second state) (third state) -1)
                     )
                 ]
                [(= action 3)
                 (if (and (validate_move row (- column 1)) (move_has_no_wall
row column (- column 1)))
                     (list (cons row (- column 1)) (second state) (third state)
-1)
                     (list (first state) (second state) (third state) -1)
                     )
                 ]
                [(= action 4)
                 (if (and (< (second state) 4) (equal-pairs (first state)
(list-ref origin_locations (second state))))
                     (list (first state) 4 (third state) -1)
                     (list (first state) (second state) (third state) -10)
                     )
                 ]
```

```
                [(= action 5)
                 (if (and (equal-pairs (first state) (third state)) (= 4
(second state)))
                     (list (first state) (second state) (third state) 20)
                     (list (first state) (second state) (third state) -10)
                     )
                 ]
                ))

  (append ans (list (= (fourth ans))))
  )

(define (Q_handler Q state action [new_value (void)])
  (define action_Q (vector-ref (vector-ref (vector-ref (vector-ref Q (caar
state)) (cdar state)) (second state)) (third state)))
  (if (void? new_value)
      (vector-ref action_Q action)
      (begin
        (vector-set! action_Q action new_value)
        Q
        )
      )
  )

(define (run_Q_learning)
  (define episodes 10000)
  (define epsilon 0.8)
  (define epsilon_decay_value 0.00008)
  (define learning_rate 0.2)
  (define discount 0.9)

  (define Q (build-vector 5
                          (lambda (i)
                            (build-vector 5 (lambda (j)
                                              (build-vector 5 (lambda (k)
                                                                (build-vector 4
(lambda (l)

(make-vector 6 0)))
```

```
                                                                ))))))); (taxi
row; taxi column; passenger location; dropoff location; action)
  (define ans (q_learning episodes 200 (cons -0.5 0) taxi_environment Q
Q_handler epsilon epsilon_decay_value learning_rate discount))
  ans
  )

(run_Q_learning)
```

Chapter 4

Conclusion

The thesis attempted to understand how functional programming could be applied to solve reinforcement learning problems in a way that the advantages of this programming paradigm could be noted. The execution of the thesis was via a library that solved Q learning problems in racket using functional programming and also incentivizing functional programming to be used when applying the library.
We can see clear advantages over using pure Racket to solve the problems and we can also note that the code has a modularity that makes it easily testable and to understand.

Future work

The thesis only approached solving reinforcement learning problems via Q Learning, without attempting to allow solutions using other algorithms such as SARSA or DQNs. In future work, the library could be extended to include this algorithms and provide a complete set of functions to solve reinforcement learning problems using Racket, being the first library of its kind.
An alternative roadmap would be to turn the library into a complete DSL which would allow to build even simpler solutions to the problems.

Bibliography

- Racket. (n.d.). Retrieved February 11, 2022, from https://racket-lang.org/
- Butterick, M. (n.d.). Beautiful racket: an introduction to language-oriented programming using racket. Retrieved from https://beautifulracket.com/.

- Amershi, S. (n.d.). Software Engineering for Machine Learning: A Case study. https://doi.org/https://www.microsoft.com/en-us/research/uploads/prod/2019/03/amershi-icse-2019_Software_Engineering_for_Machine_Learning.pdf
- *Open AI Gym documentation*. Gym Documentation. (n.d.). Retrieved February 11, 2022, from https://www.gymlibrary.ml/