APRIL 19, 2019

Q-LEARNING DYNAMIC PROGRAMMING FOR WORLD GRID NAVIGATION

REINFORCEMENT LEARNING PROJECT

SAMYUEL DANYO

https://github.com/SamyuelDanyo
https://www.linkedin.com/in/samyueldanyo/

Introduction

In this report I present my implementation of the **Q-L**earning (**QL**) algorithm, which is from the family of **R**einforcement **L**earning (**RL**) algorithms. The implementation is specifically designed for solving a world grid navigation problem, utilizing either QL or **D**ynamic **P**rogramming (**DP**). The method is a full-custom implementation in Python. Additionally, visualizations, performance results and analysis of a world grid navigation experiment are discussed.

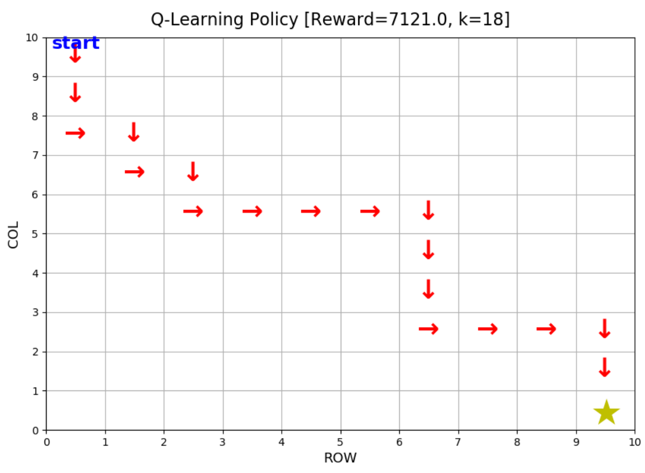
The goal of the project is to study the effectiveness of QL to dynamically learn, as well as, the effects of the system's configuration & the input reward topology on performance.

The target world grid navigation problem to is defined as follows: The world is a 10x10 grid, with 100 states total. The agent starts at state 0 (the idle state), attempting to reach state 100 (the goal state). In order to guide the agent, a reward is given based on its state and action. The reward is loaded from task1.mat, and can be seen in the exported jupyter notebook (NN_Project_2.html).

Four different parameter decay functions are implemented, namely: k_decay, const_k_decay, log_k_decay, const_log_k_decay.

For performance analysis of the learnt Q-Function, a few different metrics are observed: success in deducting an optimal policy for reaching the goal state (success run rate), training time, success in reaching the goal state during learning (success trial rate), optimal policy reward, optimal policy number of actions, exploration/exploitation ratio.

While the main metrics for evaluating the performance of the method are: the successful learning of the optimal Q-Function distribution, as well as, optimal policy reward & action sequence length, observing execution time & exploration/exploitation ratio gives interesting insights into how the learning parameters' configuration affects the learning process and hence how is that reflected into the resultant performance.



These effects are studied through running the QL algorithm with various learning parameters configurations, mixing the above-mentioned epsilon/learning rate decay functions with reward discount factors in [0.5;0.9]. Each configuration was run 10 times in order for its success rate in extracting an optimal policy to be found.

The exploration/exploitation ratio can be directly linked to the ability of the configuration to yield an optimal policy. The effects of the decay rate and reward discounts on the ratio are also analyzed.

The algorithm is implemented mainly through using NumPy for matrices manipulation and calculations. Helper functions are implemented for parameter decay, evaluation metrics extraction, as well as, wrapped training and displaying results using matplotlib for plotting graphs and some other basic helper packages.

Environment Class

Design

The class implements the environment functionality in the QL algorithm. It is responsible for tracking the agent location (state) and performing the agent steps, by computing the next state and issuing reward.

Implementation

The Class

The environment is implemented as a class with ten fields: {`state_grid_dim, goal_state, idle_state, state, obstacle_state, action_dim, action_dict, action_coords, action_trans_conf, R_table`} for defining the state space, action space and reward table. And five methods:

- `__init__(self, state_grid_dim, goal_state, R_table, idle_state, action_trans_conf):`
  - For creating the object & declaring its fields.
- `reset(self):`
  - For resetting the agent state.
- `step(self, action):`
  - For calculating the next state, reward and success based on the agent's action.
- `allowed_actions(self)`
  - For generating a list of valid actions, based on the agent's location.
- `_build_reward_table(self):`
  - For building a reward table if such is not provided.

Agent Class

Design

The class implements the agent functionality in the QL algorithm. It is responsible for making decisions (taking actions), utilizing the epsilon-greedy approach, based on epsilon, current Q-Function and current state (valid actions).

Implementation

The Algorithm Class

The agent is implemented as a class with no fields {} and two methods:

- `__init__(self):`
  - For creating the object.
- `get_action(self, env, Q, epsilon,explore_count,exploit_count):`
  - For deciding on an action, implementing the designed, mentioned above.

Q-Learning Class

Design

Q-Learning is an unsupervised Machine Learning approach from the family of Reinforcement Learning algorithms. It deals with uncertain and usually unknown system by utilizing trial and error through exploration and incentive in the form of reward.

The QL system is comprised of an environment and an agent. The objective is for the agent to maximize the received reward while performing a given task.

Q-Learning is based on a few ideas:

Markov **D**ecision **P**rocess (**MDP**): is a model for a dynamic system which exhibits the so-called Markov property: [given the present, the future does not depend on the past.]

MDP comprises of a few elements:

- Set of states: S = {s1; s2: s}
- Set of actions: A = {a1; a2: a}
- (State) Transition Function:
    - Deterministic – f: S x A => S
    - Non-deterministic (Probability of the system reaching the new state ***s'*** after taking action ***a*** at state ***s***) – f: S x A x S => [0:1]
- Reward & Return: When the agent takes an action ***a*** to make a transition from state ***s*** to next state ***s'***, reward ***r*** is assigned by the environment. The agent receives the reward in the next state ***s'***.
    - Reward Function: S x A x S => R
    - Total Reward (Return): The total reward expected in the future by doing k-steps (transitions), with future rewards being discounted by a factor of ***y***.
        - $\mathbf{R}_0 = \mathbf{r}_1 + \mathbf{y}*\mathbf{r}_2 + \mathbf{y}\text{^}2*\mathbf{r}_3 \ldots + \mathbf{y}\text{^}(k-1)*\mathbf{r}_k$

Controlling MDP by imposing a decision-making mechanism - Policy:

- Deterministic (Based on a state, an action is taken).
    - n: S => A
- Non-deterministic (Based on a state and action a probability is given).
    - n: S x A => [0:1]

Q-**F**unction (**QF**): Measures the “worth” of taking a specific action ***a*** at a particular state ***s*** under a given policy ***n*** based on the expected Return.

- Qn: S x A => R

Optimal Policy: Maximizes (with respect to a given task) the QF over all possible (***s***,***a***) pairs.

Bellman Equation: Provides an iterative method for determining the values of the Q-function for a given policy by defining a direct relationship between the value of QF[$\boldsymbol{s}_0,\boldsymbol{a}_0$] and the value of QF[$\boldsymbol{s}_1,\boldsymbol{a}_1$]

- $\mathbf{Q}_n[\mathbf{s},\mathbf{a}] = \text{ADD}(\text{Prob}[\mathbf{s},\mathbf{a},\mathbf{s'}] * (\mathbf{r}[\mathbf{s},\mathbf{a},\mathbf{s'}] + \mathbf{y}*\mathbf{Q}_n[\mathbf{s'},\mathbf{a'}]))$

Dynamic Programming: Defines an iterative method for calculating the Q-Function, using the Bellman Equation. It needs the State Transition Model (STM) of the system (what actions in each state can the agent make & their probabilities if non-deterministic).

- Bellman Optimality Equation & Principle of Optimality: The optimal value of:
  - **Q**[**s,a**] = ADD(Prob[**s,a,s'**] * (**r**[**s,a,s'**] + **y***max(**Q**n[**s',a'**])))
- Optimal policy has the property that whatever the initial state and initial action are, the remaining actions must constitute an optimal policy with regards to the state resulting from the first action.
- Value Iteration Algorithm:

```
In:  STM, R, y, Q=0
    while (Qnew – Qold) != 0:
        for each Q[s,a]:
            Q[s,a] = ADD(Prob[s,a,s'] * (r[s,a,s'] + y*max(Qn[s',a'])))
```

Q-Learning defines an approach to finding the optimal Q-Function values without having a prior STM. It learns the Q-Function (rather than calculating it as Dynamic Programming does) on the basis of observed states only and the corresponding received reward:

- **Q**[**s,a**]' = **Q**[**s,a**] + **lr** * (**r**[**s,a,s'**] + **y***max(**Q**[**s',a'**]) - **Q**[**s,a**])

It is proved that the learning converges with **k** -> ***inf***, if all [**s,a**] have been ***inf*** often chosen. In real life though - QL can often get stuck depending on the reward topology and learning configuration.

- Exploration & Exploitation: Is a method which deals with trying to converge the QL.
  - Exploitation is when the agent picks the action, which provides it with the max return (max Q-value). Exploitation is needed so the agent uses the QF it has already learnt in order to extract optimal policy and max reward. The problem is the current QF is based on the agent's observations. If they are not close to the "truth", it will get stuck into a "local max" of the reward or never accomplishing its given goal.
- Exploration is when the agent picks on random action other than the optimal one. Exploration is needed so the agent can learn the QF distribution and hence be able to find an optimal policy which reflects the real truth.
- Epsilon-Greedy Exploration: is an approach to balance exploration & exploitation:

  a = argmax(**Q**[**s,a**]) Probability = **1 – epsilon** (**e**)
  - **a** is

  random_choice(**a** != argmax(**Q**[**s,a**])) Probability = **epsilon** (**e**)

- Q-Learning Algorithm (e-Greedy Exploration):

```
In:  e, R, lr, y, Q=0, s0
    while convergence or trials>N:
        while success or lr<m:
            a = e-Greedy_Exploration()
            s', R[s,a], success = env()
            Q[s,a]' += lr * (R[s,a] + y*max(Q[s',a']) - Q[s,a])
            s=s'
```

Implementation

The Class

The Q-Learning is implemented as a class with ten fields {`env, agent, mode, ST_TR_M, epsilon_fn, epsilon, LR_fn, LR, R_discount, Q`} for defining the environment, agent, mode and learning configuration.
As well as, seven methods:

- ❖ `__init__(self, state_grid_dim, goal_state, R_table, idle_state, epsilon_fn, LR_fn, R_discount, mode, ST_TR_M, action_trans_conf):`
  - For creating the object & declaring its fields.
- ❖ `update(self, memory):`
  - For updating the Q-function map utilizing the Bellman Equation.
- ❖ `train(self, verbose=False):`
  - For performing Q-Learning, implementing the Q-Learning algorithm.
- ❖ `dynamic_programming(self):`
  - For performing Dynamic Programming, implementing the Value Iteration algorithm.
- ❖ `_build_state_transition_model(self):`
  - For building simple deterministic state transition model.
- ❖ `get_optimal_policy(self):`
  - For getting greedy policy; optimal policy.
- ❖ `display_policy(self, policy,policy_coord,reward,greedy_policy):`
  - For plotting optimal policy and displaying greedy policy.

The Helper Functions

The QL algorithm class has seven helper function:

- ❖ `perform_QL(state_grid_dim, goal_state, R_table, idle_state, epsilon_fn, LR_fn, R_discount, mode, ST_TR_M, action_trans_conf, max_nb_trials, verbose):`
  - For wrapping QL/DP, data gathering & display.
- ❖ `k_decay(par=1, k=1):`
- ❖ `const_k_decay(par=1, k=1, const=100):`
- ❖ `log_k_decay(par=1, k=1):`
- ❖ `const_log_k_decay(par=1, k=1, const=5):`
- ❖ `const(par=1, k=1, const=0.99):`
- ❖ `const_decay(epsilon=1, k=1, decay_rate=0.99):`

Experimental Setup

Q-Learning testing is performed for each of the learning configurations: parameter decay functions – (epsilon_fn, LR_fn) ∈ {k_decay, const_k_decay, log_k_decay, const_log_k_decay}; reward discount factor – R_discount ∈ {0.5, 0.9}; with reward=task1.mat. Q-Learning evaluation is performed with (epsilon_fn, LR_fn) = const_k_decay; R_discount = 0.9; with reward=qeval.mat. Additionally, Dynamic Programming is utilized for comparison purposes.

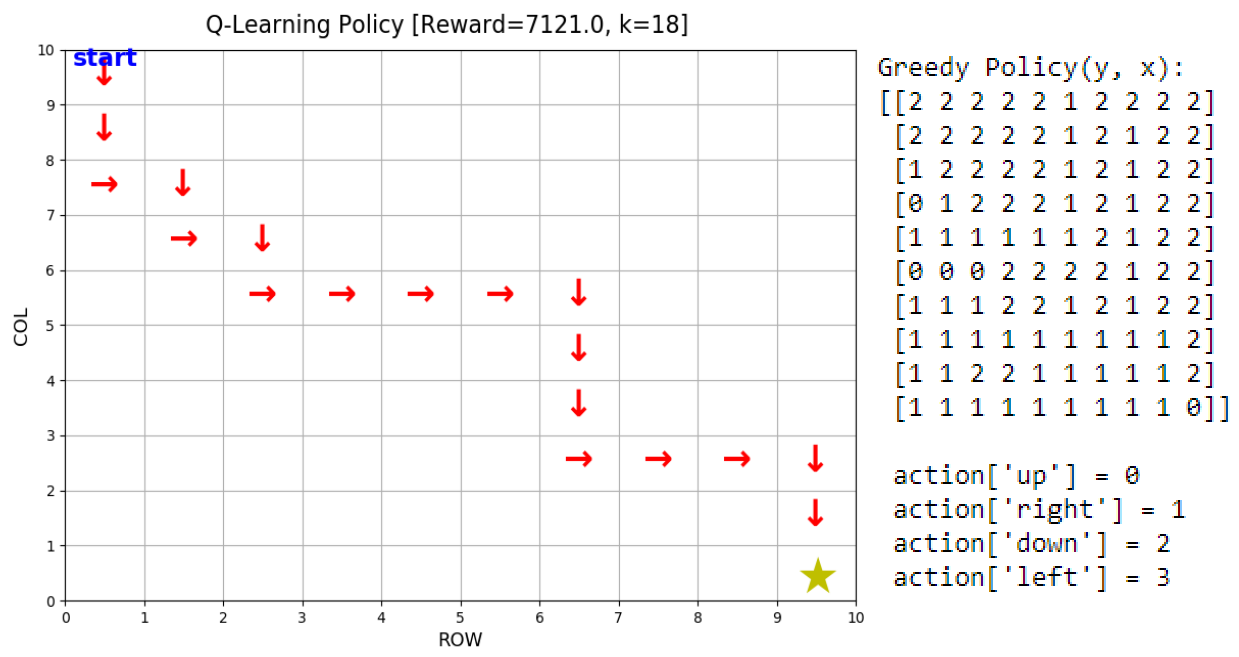
- ❖ Q-Learn & Test | all configurations, each ran for 10 times.

- ❖ Dynamic Programming & Test | utilizing reward=task1.mat, auto-generated State-Transition Model.
- ❖ Q-Learn & Evaluation | the above-mentioned evaluation configuration.
- ❖ Dynamic Programming & Evaluation | utilizing reward=qeval.mat, auto-generated State-Transition Model.

Results and Discussion

From the QL performance results (Table 1), it can be seen that three out of the total eight learning configurations, namely: const_k_decay + y=0.9, log_k_decay + y=0.9 and const_log_k_decay + y=0.9, lead to convergence and achieving the goal state. The const_k_decay shows a clear outperformance with an execution time ~2x less than const_log_k_decay. It is interesting to note that while the two parameter decay functions in combination with y=0.9 have a 100% success run rate, the same with y=0.5 have 0% success run rate.

The optimal policy is displayed below as of the path taken by the robot and also as a greedy policy of the whole world. The optimal reward is 7121, achieved with 18 actions. It was derived by QL and confirmed by Dynamic Programming, which proved that the Q-learning experiment is able to converge to an optimal solution.



```
Greedy Policy(y, x):
[[2 2 2 2 2 1 2 2 2 2]
 [2 2 2 2 2 1 2 1 2 2]
 [1 2 2 2 2 1 2 1 2 2]
 [0 1 2 2 2 1 2 1 2 2]
 [1 1 1 1 1 1 2 1 2 2]
 [0 0 0 2 2 2 2 1 2 2]
 [1 1 1 2 2 1 2 1 2 2]
 [1 1 1 1 1 1 1 1 1 2]
 [1 1 2 2 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 0]]

action['up'] = 0
action['right'] = 1
action['down'] = 2
action['left'] = 3
```

Fig. 1: Optimal Policy.

Effects of Parameter Decay Function Choice

When looking at the QL performance results (Table 1) below, it can be clearly observed that the choice of how epsilon (the exploration probability) and learning rate are changed in time has a significant effect on the QL performance.

- ❖ k_decay has the lowest execution time, since the parameters decay the fastest. The parameters' decay is linear, meaning that epsilon declines extremely fast to negligent values, which prevents the algorithm from exploring the world and the effect of this is shown by the corresponding success run rate of 0%. This can be chased deeper by looking at the exploration/exploitation section of Table 2, below. It can be seen that the exploration/exploitation ratio of action taken by the agent is only ~3%. Furthermore, the learning rate would plummet, too. Based on that it can be deducted that the learnt Q-Function will be far from the truth (optimal QF) and hence the agent would get stuck in between a local reward maximum.

Table 1: Q-Learning Performance. Gold=best.

PARAMETER VALUES AND PERFORMANCE OF Q-LEARNING				
ε_k, α_k	No. of Goal-Reached Runs		Execution Time (sec.) [Success Average, Total Average]	
	y=0.5	y=0.9	y=0.5	y=0.9
1/k	0/10 0.00%	0/10 0.00%	0.00 37.76	0.00 37.49
100/100+k	0/10 0.00%	10/10 100.00%	0.00 47.18	37.84 37.84
(1+log(k))/k	0/10 0.00%	5/10 50.00%	0.00 297.42	75.54 234.84
(1+5*log(k))/k	0/10 0.00%	10/10 100.00%	0.00 448.61	113.45 113.45

- Const_k_decay proves to be the best choice out of the approached parameter decay functions. As seen by Table 1., it is able to converge with 100% success run rate and at optimal execution time. This is due to the better epsilon and learning rate decay. Taking the constant=100, the epsilon and learning rate decay would be slow for the first ~200 k, meaning that the agent will be allowed to explore and learn more. This is observed in Table 2, with const_k_decay having the highest exploration/exploitation rate of ~50%. The learnt Q-Function would be a good representation of the truth and hence using a greedy policy would converge to the optimal policy.
- Log_k_decay provides very interesting insights. With y=0.9, it achieves 50% success run rate at a significantly higher execution time than either of the previous decay functions. This is due to the parameters decaying faster when k is small and slower with k growing. When looking at Table 2, this is confirmed by it having the lowest exploration/exploitation rate of ~2.3%. However, with contrast to k_decay, it is able to converge 5 out of 10 times. This is due to the fact that while it learns slower, it continues doing so for a longer period of time. It can be seen that the success trial rate is significantly lower that the const_k_decay (~32% vs ~99%), which is to be expected due to the low exploration. While due to the long learning it is sometimes able to converge to a solution, it can be seen (from Table 2, Reward, k, Success section), that the path taken is not the optimal and it results in reward=6860, compared to the optimal 7121.
- Having in mind the above-made deductions, it is easy to infer what to expect from const_log_k_decay. It holds the parameter values higher then log_k_decay for small k and it also decays slowly. This, as shown by the data in Table 2, results in higher exploration/exploitation ratio and also longer execution time (and hence learning).

Thanks to that it has a very high success trial rate and it also achieves 100% success run rate for y=0.9 with optimal trajectory (reward=7121, k=18).

Table 2: Q-Learning Exploration/Exploitation & Success.

EXPLORATION/EXPLOITATION & SUCCESS OF Q-LEARNING				
ε_k, α_k	**Exploration/Exploitation**		**Trial Success Rate, Reward, k, Success/Fail**	
	y=0.5	**y=0.9**	**y=0.5**	**y=0.9**

1/k	17784/585216
	3.04%
		17653/585347
		3.02%
			0/3000 (0.00%)
			R=13990
			k=101
				0/3000 (0.00%)
				R=13990
				k=101

100/100+k	279744/499146
	43.34%
		223667/381749
		58.59%
			2988/3000 (99.60%)
			R=13990
			k=101
				2974/3000 (99.13%)
				R=7121
				k=18

(1+log(k))/k	91530/3691187
	2.48%
		106694/4954306
		2.15%
			931/3000 (31.03%)
			R=13990
			k=101
				972/3000 (32.40%)
				R=6860
				k=18

(1+5*log(k))/k	252166/5001291
	5.04%
		141182/1100335
		12.83%
			2515/3000 (83.83%)
			R=13990
			k=101
				2887/3000 (96.23%)
				R=7121
				k=18

Effects of Reward Discount Factor Choice

As by theory, higher reward discount factor (future rewards being discounted LESS), results in the agent making greedy decisions with “longer-term” perspective. Lower y, results in the opposite – the agent makes greedy decision based on the immediate reward.

The benefit of each approach is very much dependent on the nature of the reward function. If the reward “guides” the agent based on immediate incentive – lower y would be more beneficial, in the opposite case – higher y would outperform.

In general, using lower y introduces the risk of getting stuck in local maxima, while higher y, can make the learning harder by propagating errors and causing confusion (instability).

With the tested reward (task1.mat), it can be seen that y=0.9 provides better convergence. This is thanks to, as per theory, it preventing the agent to get stuck, where the immediate reward is at a local maximum, but the extended future return (with respect to the goal) is not. The effect of y, is huge. In the case of const_k_decay it makes the difference between 0% and 100% success run rate. Even though, as discussed, the agent has over 99% success trial rate for const_k_decay + y=0.5 thanks to the high exploration/exploitation ratio, this amounts to 0% success run rate because of getting stuck in local maxima. The same deduction can be made for all other configurations with y=0.5

Final Discussion

Dynamic Programming is used to calculate the optimal policy and confirm results and observations from the Q-learning experiment. It proves that the optimal Q-Function is learned by the algorithm and results in: reward=7121, k=18. As expected in all cases, it can be seen that successful run times are smaller than the average (and hence non-successful run times). This is to be expected as when the goal state is reached, the trial is stopped.

In the experiment, const_k_decay, log_k_decay and const_log_k_decay parameter functions were able to achieve full or partial convergence, with const_k_decay and const_log_k_decay chieving optimal convergence. K_decay was unable to converge and log_k_decay achieved non-optimal convergence.

The outperformance of const_k_decay and const_log_k_decay are due to them allowing the agent to explore more, which results in a better optimal Q-Function representation. While cosnt_log_k_decay has ~3 times smaller exploration/exploitation ratio than const_k_decay, it overall decays slower (for larger k), and hence allows comparative amount of explorations at the cost of longer execution time. This might prove advantages if the constant of const_k_decay is too small compared to the world size and hence while it explores more as a percentage of actions taken, it does not do so in absolute terms. In such a case (small const or large world) I would expect const_log_k_decay to outperform.

For the presented test case, the results suggest const_k_decay to be the best parameter decay function, since it achieved optimal convergence with 100% success run rate ~3 times faster than const_log_k_decay.

As of reward discount factor – 0.9 is without question the better choice for the particular problem since it incentivizes the agent to move towards the true maximum expected return, preventing it from getting stuck in local maxima. In general, the value of y should be carefully tuned relative to the reward topology. A dynamic reward discount factor, which grows in time might prove to outperform.

All in all, 3 out of the 8 configurations converged to reach the goal state, all with reward discount factor = 0.9. Despite successful trials due to good exploration, all configurations with reward factor = 0.5 got stuck in the same local maxima as deducted by Table 2's reward and k section. K_decay was unable to converge at all, due to low exploration and learning because of the fast, linear decay of epsilon and learning rate. Const_k_decay and log_k_decay converged to optimal policy (reward=7121, k=18) with 100% success run rate and over 99% success trial rate thanks to good exploration and learning activity.

Since const_k_decay + y=0.9 proved to be the optimal configuration, it will be used to perform the evaluation. Since the constant=100 and trial termination at 0.005, gives over 200 actions before termination, I believe it will be sufficient to explore the 10x10 world grid well, while learning Q, while keeping the execution time low.