Reinforced Machine Learning using Q-Algorithm
Documentation

Author:Stiliyan Tonev
Contact:stiliyan21[at]gmail.com

Content:

Abstract

Reinforced learning is a type of inteligent computer agent, which has the ability to observe and adapt to the enviroment he is put in. Here is an implementation of the Q-Algotithm which is used for such agents. It is not the only one but it is a steping stone for other, rival algorithms. This implementation was build with generality in mind, so the developer can make multy-dimensional enviroment without too much effort.

What is Q-Learning?

Q-Learning is a type of reinforced learning algorithm. Reinforced means it observes and adapts to the enviroment. It is used in Artificial Inteligence (AI) systems in order to simulate human like process of learning. It is related with the so-called “Markov Chains”. They represent the math side of the algorithm, but in many places it is described by logic and is quite intuitive, which makes it easyer and more understandable for begginers.

How can it be used?

The algorithm finds application where the “agent” is left unsupervided and is expected to reach a given conclusion.
For example a robot, of those that clean the floor, during the procces of cleaning it can bump in to an obstacle, or reach a point from which it can’t return (stairs), so by using the Q-Algorithm the robot will learn that those places are dangerous and won’t pass throu them anymore.

Another example can be creating a picture by a given model, the “agent” will learn about the given model in the first few tryes to recreate it, collecting information about the models general properties, and later use that information to generate some thing which follows those general properties.

It is also used in order to find paths, as with every new cycle it learns which path is better than the previous, and after learning enough it will be able to determine the optimal path to a given objective.

Learning how it works.

The structure of the algorithm is as follows:

- S → States in which the “agent” can go, those who are forbiden too, each state has a “reward”.
- A → Actions which the “agent” can take, each state has a set of actions the agent can take, and each action has a value, and an exit state.
- Learning coeficent (α) [0,1] → It plays a role like a believe factor, if it is 1 or larger the “agent” will believe he is on the right way evin if he is not.
- Discount Factor (γ) [0,1] → Tells the agent how much will the next action matter, it can also lead to the wrong path if too high.
- s → Current state of the “agent”.
- a → Action choosed from the current state.
- s’ (next state)→ The state in which is the “agent” after executing action “a” in state “s”.
- a’ (next action) → The choosed action in state s’.

And the o-so-holy Q-Function:

- $Q(s,a) = r + \alpha*\gamma*(\max_{a'}(s'))$
- Which translated means “the Quality of action ‘a’ from state ‘s’ is equal to the (r)Reward in state s’ plus (learning coeficent)*(discount)*(the action with most value from s’)”
- Simply said follow the bigger value.

Now as we have “undurstood” the math lets look at how it works with the enviroment.

We will work it out on a 3x3 grid. Don’t worry if you are confused it will get clearer the more you read it.

0.1111	1.0	-1.0
0.5	0.0	-1.0
2.0	0.3	0.0

α=0.8 γ=0.5

Now our agent start from the top-left cell (0.1111). It chooses the action by comparing the actions value, for our conviniance we will use the reward of the next state as action value so,

Right: 1.0 Down:0.5

It chooses the larger one (Right), and updates the transition value [Right] = Q(top-left,right) = 1.0(reward)+0.8*0.5*(max(0.0,0.1111,-1.0)) = 1.0+0.8*0.5*0 = 1+0.40 = 1.40

And now from the top-center cell:

Down: 0.0 Right:-1.0

Left:(if you go back there you enter endless cycle)

Now the largest is 0.0:

[Down] = Q(top-center,down) = 0.0+0.8*0.5*(max(0.5,0.3,0.0)) = 1.0+0.8*0.5*0.5 = 1+0.20 = 1.20

And from there it must be clear that we will end up in the bottom-left cell, because we will move to the biggest neightbour (0.5) and the next move with most value is (2.0).

Indeed this is a very rought example, but the thing you must remember/understand:

{choose action,estimate Q(s,a),assign to the present action,move to s’,repeat}

As we said before that Q in Q-Learning is for Quality which is estimated by own, and neightbour value.

My implementation and how to make it run

My implementation is a java package, containing 3 main files.

1. Mind
2. States
3. Actions

The file Actions.java is the shortest and most simple one.
It contains the declaration of “Actions” theyr property “Reward” (Value), Name, and EndState.
The value is the “Reward” you get after executing this action.
The end state can be different or remain the same.
And the name is used to make it simpler and understandable.

(Action:
        Name=”Go to university”
        Reward=0.001
        EndState=”University”
Action:
        Name=”Get coffee”
        Reward=2.999
        EndState=BegginState
)

It can be constructed without arguments and using the setters assing values to, or called with its different constructors.
Properties:

- String   Name;

- double Reward;
- States EndState;

Constructors:

- Actions();
- Actions(EndState);
- Actions(name,endstate);

Methods:

- void setName(”Other action name”);
- String getName()
- double getReward();
- void setReward(1.0);
- void setState(endstate);
- States getState();

States.java is a little bit more complex file (but wait till we get to mind).
Every state has “Attributes” which by type is a HashMap → [String,Double]
Properties:

- List<Actions> Actions; → Actions that can be taken from this state.
- HashMap<String,Double> Attributes; → Allows the developer to set different (custom) attributes to actions. Like you want a “checked” marker to keep track of checked cells just add it in to the HashMap and use it when printing. Use youre imagination.

Constructors:

- States();
- States(List<String> str,List<Double> arg); → Adds them to the HashMap as each string coresponds to the same index in arg.
- States();

Methods:

- Actions getBestAction(); → Returns the action with most value.
- Actions getImprovedAction(); → Same as getBeastAction() except it omits actions that will lead to unsatisfaing states no mater they'r value.
- void setAttribute("attribute name",value) → Sets the value of the given attribute to the given.
- double Use("attribute name") → Returns the value of the attribute with the given name.
- void setGoal() → Sets the "exit" attribute to 1, there fore this is an exit state and is satisfaing one.
- void setObstacle() → Sets the "exit" attribute to -1, there fore an exit state but not a satisfaing one.