Deep Reinforcement Learning for Maze Solving

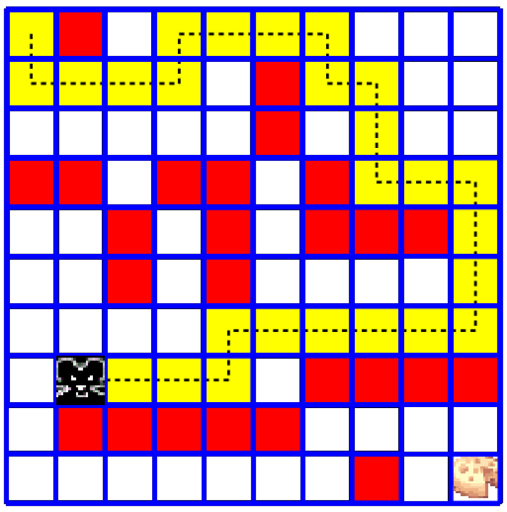
Deep Reinforcement Learning got a lot of publicity recently due to Google's acquired AI Startup DeepMind For More than 500M$ and Intel's 15B$ Mobileye deal. Since then, Google has also patented the mathematical aparatus of using Q-learning with deep neural networks that was developed by its **DeepMind** division. In this notebook we will try to explain the main ideas behind deep reinforcement learning (also called deep **Q-learning**) by a simple application for solving classical mazes.

Background

Reinforcement learning is a machine learning technique for solving problems by a feedback system (rewards and penalties) applied on an **agent** which operates in an **environment** and needs to move through a series of states in order to reach a pre-defined final state. A classical example is a rat (agent) which is trying to find the shortest route from a starting cell to a target cheese cell in a maze (environment). The agent is **experimenting** and **exploiting** past experiences (**episodes**) in order to achieve its goal. It may fail again and again, but hopefully, after lots of trial and error (rewards and penalties) it will arrive to the solution of the problem. The solution will be reached if the agent finds the optimal sequence of states in which the **accumulated sum of rewards** is maximal (in short, we lure the agent to accumulate a maximal reward, and while doing so, he actually solves our problem). Note that it may happen that in order to reach the goal, the agent will have to endure many penalties (negative rewards) on its way. For example, the rat in the above maze gets a small penalty for every legal move. The reason for that is that we want it to get to the target cell in the shortest possible path. However, the shortest path to the target cheese cell is sometimes long and winding, and our agent (the rat) may have to endure many penalties until he gets to the "cheese" (sometimes called "delayed reward").

Maze Solving:

Traditional maze puzzles have been used a lot in data structures and algorithms research and education. The well-known **Dijkstra shortest path algorithm** is still the most practical method for solving such puzzles, but due to their familiarity and intuititive nature, these puzzles are quite good for demonstrating and testing Reiforcement Learning techniques.



Exploitation vs. Exploration:

We already explained what **min_reward** and **actions_dict** are. The story of **epsilon**, also called **exploration factor** is part of the larger **Q-learning** story:

The main objective of Q-training is to develope a **policy** π for navigating the maze successfully. Presumably, after playing hundreds of games, the **agent** (rat in our case) should attain a clear deterministic policy for how to act in every possible situation. Which action to take in every possible maze state?

The term **policy** should be understood simply as a function πthat takes a maze snapshot (**envstate**) as input and returns the action to be taken by the **agent** (rat in our case). The input consists of the full nxn maze cells state and the rat location.

$$\textbf{next action}=\pi(\textbf{state})$$

At start, we simply choose a completely random policy. Then we use it to play thousands of gamesfrom which we learn how to perfect it. Surely, at the early training stages, our policy πwill yield lots of errors and cause us to lose many games, but our rewarding policy should provide feedback for it on how to improve itself. Our learning engine is going to be a simple feed-forward neural network which takes an environment state (maze cells) as input and yields a reward per action vector (see later for better description).In order to enhance the Q-learning process, we shall use two types of moves:

Exploitation: these are moves that our policy πdictates based on previous experiences. The policy function is used in about 90% of the moves before it is completed.

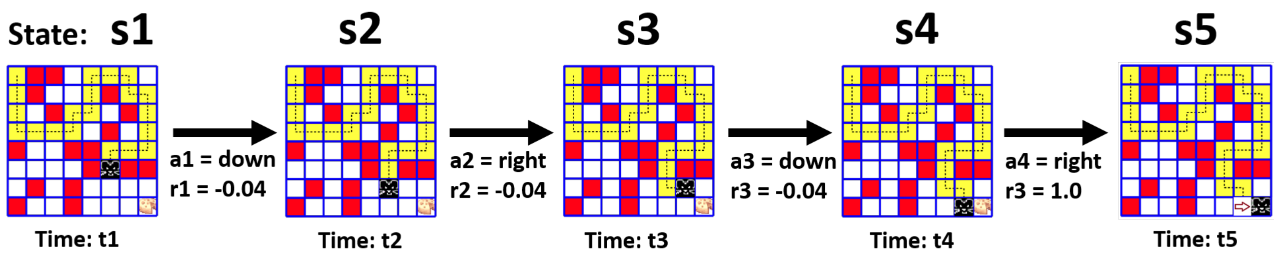
Exploration: in about 10% of the cases, we take a completely random action in order to acquire new experiences (and possibly meet bigger rewards) which our strategy function may not allow us to make due to its restrictive nature. Think of it as choosing a completey random new restaurant once in a while instead of choosing the routine restaurants that you already familiar with. The exploration factor **epsilon** is the the frequency level of how much **exploration** to do. It is usually set to 0.1, which roughly means that in one of every 10 moves the agent takes a completely random action. There are however many other usage schemes you can try (you can even tune epsilon during training!)

Markov Decision Process for Maze Environment:

A Reinforcement Learning system consists of an **environment** and a dynamic **agent** which acts in this environment in finite discrete list of time steps.
At every time step t, the agent is entering a state s, and needs to choose an action a from a fixed set of possible actions. The decision about which action to take should depend on the current state only (previous actions history is irrelevant). This is sometimes reffered to as **MDP: Markov Decision Process** (or shortly Markov Chain).
The result of performing action a at time t will result in a transition from a current state s at time t to a new state $s'=T(s,a)$ at time $t+1$, and an immediate reward $r=R(s,a)$ (numerical value) which is collected by the agent after each action (could be called a "penalty" in case the reward is negative). T is usually calle the **transition function**, and R is the **reward function**:

$$s' r=T(s,a)=R(s,a)$$

The agent's goal is to collect the maximal total reward during a "game". The greedy policy of choosing the action that yields the highest immediate reward at state s, may not lead to the best possible total reward as it may happend that after one "lucky" strike all the subsequent moves will yield poor rewards or even penalties.
Therefore, selecting the optimal route is a real and difficult challenge (just as it is in life, delayed rewards are hard to get by).
In the following figure we see a Markov chain of 5 states of a rat in a maze game. The reward for every legal move is -0.04 which is actually a "small penalty". The reason for this is that we want to minimize the rat's route to the cheese. The more the rat wonders around and wastes time, the less reward he gets. When the rat reaches the cheese cell, he gets the maximal reward of 1.0 (all rewards are ranging from -1.0 to 1.0)



If our agent takes the action sequence (starting at state $s1$ till the game end): $a1$, $a2$, $a3$, ..., an, then the resulting total reward for this sequence is:

$$A=R(s1,a1)+R(s2,a2)+\cdots+R(sn,an)$$

Our goal is to find a **policy** function π that maps a maze state s to an "optimal" action a that we should take in order to maximize our total reward A. The policy π tells us what action to take in whatever state s we are in by simply applying it on the given state s:

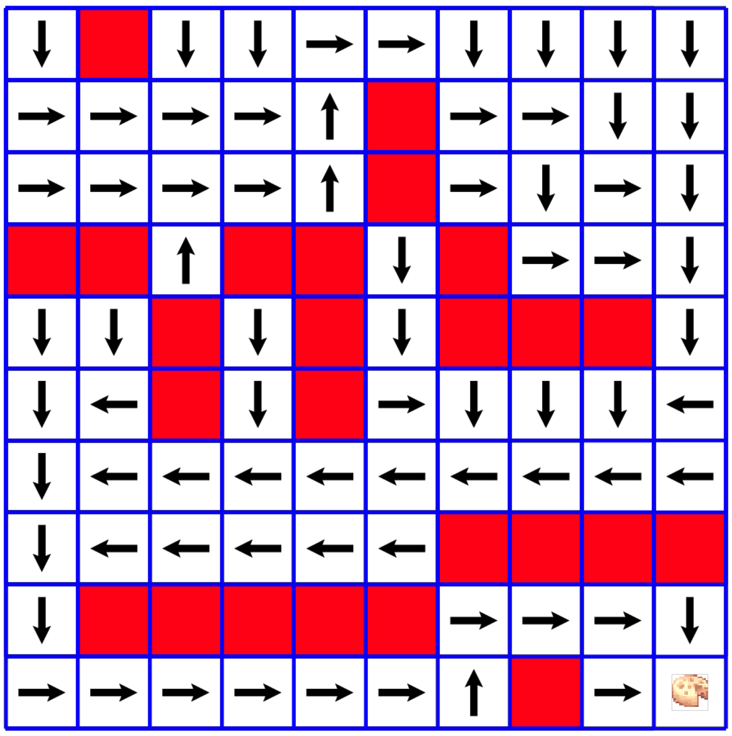
$$\text{action}=\pi(s)$$

Once we have a policy function π, all we need to do is to follow it blindly:

$$a1s2a2\cdots an=\pi(s1)=T(s1,a1)=\pi(s2)=\cdots=\pi(sn-1)$$

So playing the maze is now becoming an automatic pilot flight. We simply ask π what to do in each state and we're guaranteed to end the game with the maximal reward.
But how are we supposed to find π? From a game-theoretic point of view, it is know to be quite a difficult challenge, especially for large board games such as **GO**

A policy function is sometimes best illustrated by a policy diagram:



Q-Learning and Bellman Equation:

The trick that was used by startups such as Google DeepMind for finding π was to start with a different kind of function $Q(s,a)$ called best utility function (and sometimes best quality function, from which the Q letter and Q-learning terms were coined).
The definition of $Q(s,a)$ is simple:

$$Q(s,a)\text{=the maximum total reward we can get by choosing action } a \text{ in state } s$$

At least for our maze solving, it is easy to be convinced that such function exists, although we have no idea how to compute it efficiently (except for going through all possible Markov chains that start at state s, which is insanely inefficient). But it can also be proved mathematically for all similar Markov systems. Once we have $Q(s,a)$ at hand, finding a policy function is easy

$$\pi(s)=\mathrm{argmax}i=0,1,\ldots,n-1Q(s,ai)$$

That is: we calculate $Q(s,ai)$ for all actions ai, $i=0,1,\ldots,n-1$ (where n is the number of actions), and select the action ai for which $Q(s,ai)$ is maximal. This is obviously the way to go. But we do not have the function $Q(s,a)$ yet ... how do we get it? It turns out that the function $Q(s,a)$ has a simple recursive property which characterizes it, and also helps to approximate it. It is called Bellman's Equation and it is obvious from first sight:

$$Q(s,a)=R(s,a)+\mathrm{max}i=0,1,\ldots,n-1Q(s',ai),(\text{where } s'=T(s,a))$$

In simple words: the value $Q(s,a)$ is equal to the immediate reward $R(s,a)$ plus the maximal value of $Q(s',aj)$, where s' is the next state and ai is an action.

In addition, Bellman's Equation is also a unique characterization of the best utility function. That is, if a function $Q(s,a)$ satisfies the Bellman Equation the it must be the best utility function.

To approximate $Q(s,a)$we will build a neural network N which accepts a state s as input and outputs a vector q of q-values corresponding to our n actions: $q=(q0,q1,q2,\cdots,qn-1)$, where qi should approximate the value $Q(s,ai)$, for each action ai. Once the network is sufficiently trained and accurate, we will use it to define a policy, which we call the derived policy, as follows

$$qj\pi(s)=N[s]=\mathrm{argmax}i=0,1,\ldots,n-1(q0,q1,\ldots,qn-1)=aj$$

References:

1. Keras plays catch, a single file Reinforcement Learning example
2. Q-learning
3. Keras plays catch - code example
4. Reinforcement learning using chaotic exploration in maze world
5. A Reinforcement Learning Approach Involving a Shortest Path Finding Algorithm
6. Neural Combinatorial Optimization with Reinforcement Learning
7. Google Acquires Artificial Intelligence Startup DeepMind For More Than $500M
8. How DeepMind’s Memory Trick Helps AI Learn Faster
9. Methods and apparatus for reinforcement learning US 20150100530 A1 (Patent Registration)
10. Introduction to Reinforcement Learning
11. Reward function and initial values: Better choices for accelerated Goal-directed reinforcement learning
12. Andrej Karpathi blog: Deep Reinforcement Learning: Pong from Pixels