A Project Report on:

Maze Solving using Deep Reinforcement Learning

Submitted in partial fulfillment of requirment for the award of degree of:

Bachelor of Engineering

in

Computer Engineering

by

L.Aniruth Naraayanan(16102005)
Siddhesh Kokane(16102054)

Under the Guidance of:

Jaya D Gupta

Department of Computer Engineering
A.P. Shah Institute of Technology
G.B.Road, Kasarvadavli, Thane(W), Mumbai-400615
UNIVERSITY OF MUMBAI
Academic Year 2017-2018

Approval Sheet

This Project Report entitled **“Maze Solving using Deep Reinforcement Learning”** Submitted by **“L. Aniruth Naraayanan”(16102005)** and **“Siddhesh”(16102054)** is approved for the partial fulfillment of the requirement for the award of the degree of **Bachelor of Engineering in Computer Engineering** from **University of Mumbai.**

Prof. Sachin Malawe
(HOD Computer Engineering)

Jaya D. Gupta
(Project Guide)

Place: A. P. Shah Institute of Technology, Thane
Date :

Certificate

This is to certify that the project entitled **“Maze Solving using Deep ReinforcementLearning”** submitted by **“L. Aniruth Naraayanan”(16102005)** and **Siddhesh Kokane (16102054)** for the partial fulfillment of the requirement for award of a degree **Bachelor of Engineering** in **Computer Engineering** to the **University of Mumbai**,is a bonafide work carried out during academic year 2017-2018.

Prof. Sachin Malawe
(HOD Computer Engineering)

Jaya D Gupta
(Guide)

External Examiner(s):

1.

2.

Place: A. P. Shah Institute of Technology, Thane
Date :

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, We have adequately cited and referenced the original sources. We also declare that We have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission.
We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

L. Aniruth Naraayanan(16102005) Siddhesh Kokane(16102054)

Place: A. P. Shah Institute of Technology, Thane
Date :

Contents

1. Abstract

In this project we aim to create a solution to solve any MXN maze using machine learning techniques. Specifically we are using Deep Reinforcement Learning , a combination of Deep Neural Networks and Reinforced Learning .

The input maze will be a randomly generated array containgn information such as free cell, blocked cell, target etc., which will be fed to a tkinter window and plotted using matplotlib. This project willl run on a Tenserflow back-end with keras being used as the neural network library.

This project can be useful in real-world application such as path finding, mapping applications, robotics, etc..

2. Introduction

2.1. 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 intuitive nature, these puzzles are quite good for demonstrating and testing Reinforcement Learning techniques.

2.2. Reinforcement Learning:

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").

3. Maze Generation

The following presents the algorithm behind the generation of a random m x n maze.

Input :

m – no. of rows
n – no. of column

Procedure:

1. Create a m x n maze with all cells blocked – (0 in numpy array)

2. Randomly traverse the array

3. Mark every visited block as a free cell – (1 in numpy array)

4. If current cell is target cell then exit

5. Repeat IE. go to 1

Output:

Returns a numpy array containing information about all free cells, blocked , agent and target cell.

4. Maze Solving

The following procedures explain the how the generated maze is solved using reinforcement learning and neural networks.

4.1. Step 1:Feed the maze to the program from the generated input array Input :

M X N numpy array, which holds all the cells of the array and also denotes if a cell is free or not.
By default the Agent starts from top-left (0,0) and Cheese is located at bottom right(n-1,m-1) .

A framework for an **MDP** (Markov Decision Process) consists of an **environment** and an **agent** which acts in this environment. In our case the environment is a classical square maze with three types of cells:

1. Occupied Cells

2. Free Cells

3. Blocked Cells

Markov Decision Process

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). 4. 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)

Note that each state consists of all the available cells information, including the rat location. In our Python code, each state is represented by a vector of length 64 (for an 8x8 maze) with gray values 0.0 to 1.0: occupied cells is 0.0, a free cell is 1.0, and the rat cell is 0.5. History (yellow cells) is not recorded since the next move should not depend on past moves!

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:

action$=\pi(s)$. Once we have a policy function π, all we need to do is to follow it blindly:

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

Q-Learning and Bellman Equation

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.

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

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 $Q(s,a)$.

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:

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

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

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

Q-Training

The usual arrangement (as we've seen a lot) is to generate a sufficiently large dataset of (e,q) pairs,where e is an **environment state** (or maze state in our case), and $q=(q0,q1,\ldots,qn-1)$

are the correct actions q-values. To do this, we will have to simulate thousands of games and make sure that all our moves are optimal (or else our q-values may not be correct). This is of course too tedious, too hard, and impractical in most real life cases.

We will generate our training samples from using the neural network N itself, by simulating hundreds or thousands of games. We will **exploit** the **derived policy** π (from the last section) to make 90% of our game moves (the other 10% of the moves are reserved for exploration). However we will set the target function of our neural network N to be the function in the right side of Bellman's equation! Assuming that our neural network N converges, it will define a function $Q(s,a)$ which satisfies Bellman's equation, and therefore it must be the best utility function which we seek.The training of N will be done after each game move by injecting a random selection of the most recent training samples to N.

Assuming that our game skill will get better in time, we will use only a small number of the most recent training samples. We will forget old samples (which are probably bad) and will delete them from memory. In more detail: After each game move we will generate an **episode** and save it to a short term memory sequence. An **episode** is a tuple of 5 elements that we need for one training:

episode = [envstate, action, reward, envstate_next, game_over]

After each move in the game, we form this 5-elements episode and insert it to our memory sequence. In case that our memory sequence size grows beyond a fixed bound we delete elements from its tail to keep it below this bound.

The weights of network N are initialized with random values, so in the beginning N will produce awful results, but if our model parameters are chosen properly, it should converge to a solution of the Bellman Equation, and therefore later experiments are expected to be more truthful. Currently, building model that converge quickly seems to be very difficult and there is still lots of room for improvements in this issue.

The Experience Class

This is the class in which we collect our game episodes (or game experiences) within a memory list. Note that its initialization methods need to get

1. **model** - a neural network model

2. **max_memory** - maximeal length of episodes to keep. When we reach the maximal lenght of memory, each time we add a new episode, the oldest episode is deleted

3. **discount factor** - this is a special coefficient, usually denoted by γ

which is required for the Bellman equation for stochastic environments (in which state transitions are probabilistic). Here is a more practical version of the Bellman equation:

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

4.2. Step 2. Build a neural network model and train it

1. The most suitable activation function is SReLU (the S-shaped relu)

2. Our optimizer is RMSProp

3. Our loss function is mse (Mean Squared Error).

This model is saved in a json file saved and used for solving any number of mazes

Step 4. 3. Solve a maze using the model

The above generated model or a previously used model can be used to train and solve a maze. Finally plot the maze on the tkinter window to display the output

5. Technology Stack.

Programming Languages	**–**	**Python3 (v 3.5.2)**
Machine Learning Back-end	**–**	**Tensorflow**
Neural Network Library	**–**	**Keras (v 2.2.4)**
Operating System	**–**	**Ubuntu(Linux) (v 18.04 LTS)**

6. Conclusion and Scope

Thus, we have implemented an machine learning algorithm to solve a randomly generated maze.

This may not be the most efficient method of solving mazes as classical algorithm such as Dijkstra's algorithm, A star algorithm etc .. as much more efficient, but taking part in this project has given us a firm understanding of reinforcement learning and how its implementation .

This approach can be used in dynamic circumstances such as a maze with the agents and the target moving simultaneously in real time etc. and can even beat classical approaches when it comes to efficiency. Thus this project does have scope in other circumstances and cases.

Acknowledgement

We would like to thank our guide Ms. Jaya D. Gupta for guiding us throughout this project and our institute APSIT for providing us this opportunity to perform this project

References

1. Demystifying Deep Reinforcement Learning (https://www.nervanasys.com/demystifying-deep-reinforcement-learning)

2. Keras plays catch, a single file Reinforcement Learning example (http://edersantana.github.io/articles/keras_rl)

3. Q-learning (https://en.wikipedia.org/wiki/Q-learning)

4. Keras plays catch - code example

(https://gist.github.com/EderSantana/c7222daa328f0e885093)

5. Reinforcement learning using chaotic exploration in maze world

(http://ieeexplore.ieee.org/document/1491636)

6. A Reinforcement Learning Approach Involving a Shortest Path Finding Algorithm (http://incorl.hanyang.ac.kr/xe/paper/ic/ic2003-3.pdf)

7. Neural Combinatorial Optimization with Reinforcement Learning (https://arxiv.org/abs/1611.09940)

8. Google Acquires Artificial Intelligence Startup DeepMind For More Than $500M (https://techcrunch.com/2014/01/26/google-deepmind)

9. How DeepMind's Memory Trick Helps AI Learn Faster (https://www.technologyreview.com/s/603868/how-deepminds-memory-trick-helps-ai-learn-faster/?imm_mid=0ef03f&cmp=em-data-na-na-newsltr_20170320)

10. Methods and apparatus for reinforcement learning US 20150100530 A1 (Patent Registration) (https://www.google.com/patents/US20150100530)

11. Introduction to Reinforcement Learning (http://www.cs.indiana.edu/~gasser/Salsa/rl.html)

12. Reward function and initial values: Better choices for accelerated Goal-directed reinforcement learning (https://hal.archives-ouvertes.fr/hal-00331752/document)

13. Andrej Karpathi blog: Deep Reinforcement Learning: Pong from Pixels (http://karpathy.github.io/2016/05/31/rl/)