Machine Learning Engineer Nanodegree

Capstone Project

Ramtin Raji Kermani, ramtin@asu.edu, https://www.linkedin.com/in/ramtinr

May 2018

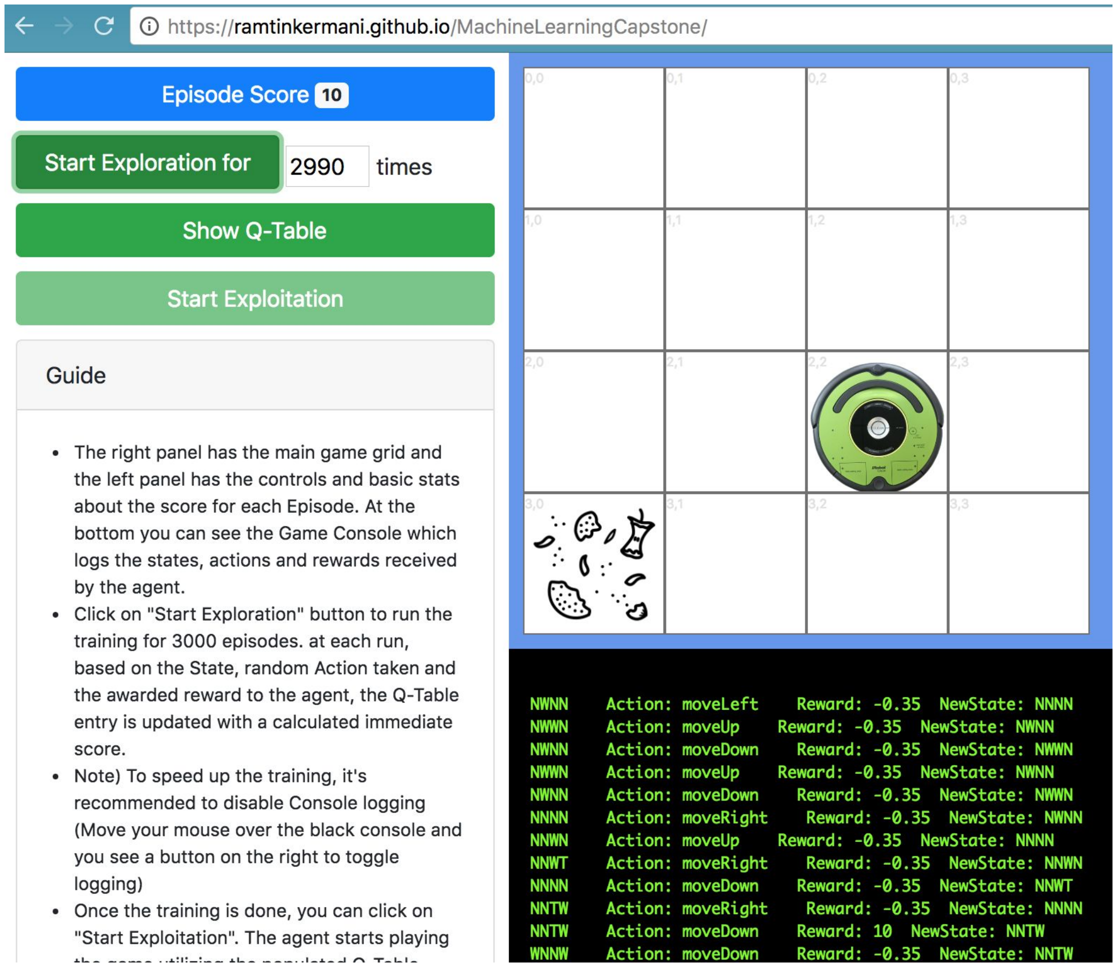
I. Definition

One of the most interesting household product families in market today are the autonomous robotic vacuum cleaners such as iRobot Roomba from iRobot corporation which is sold over 10 million unit in 2017[1] and these machines are loved by thousands. One of the main challenges of household robotic systems is the problem of accurate indoor localization and path planning. Even though these robots are called “Smart” and “Autonomous”, these machines still are far from being intelligent and efficient. They lack basic learning capabilities even though they posses various sensors to sense the environment in which they operate. Most of the sensory data is used for reactive control (avoiding a fall or hitting a wall). No matter how many times the robot runs around a small house with basic floor plan, the next time it will still bump into the couch or assumes that all areas of the house have the same amount of “dirt” to be cleaned. These issues decrease the performance of the robots as well as their life-time.

Reinforcement learning is one if the most interesting branches of machine learning that trains an agent to be **“Good”** and **“Behave nicely”** and **“Achieve goals efficiently”** by giving it a chance to explore the environment and perform actions by “**trial and error”**. The agent, based on the outcome, is either “**Punished**” or “**Rewarded**”. The rewards or punishments are awarded based on agent’s behavior, achievements and weather or not it was able to achieve its **final goal**.

Project Overview

I have created a web-based application to present a simple application of Reinforcement Learning with Q-Learning for simulated environments. The following picture shows a screenshot of the application in training phase.



Guide

- The right panel has the main game grid and the left panel has the controls and basic stats about the score for each Episode. At the bottom you can see the Game Console which logs the states, actions and rewards received by the agent.
- Click on "Start Exploration" button to run the training for 3000 episodes. at each run, based on the State, random Action taken and the awarded reward to the agent, the Q-Table entry is updated with a calculated immediate score.
- Note) To speed up the training, it's recommended to disable Console logging (Move your mouse over the black console and you see a button on the right to toggle logging)
- Once the training is done, you can click on "Start Exploitation". The agent starts playing

```
NWNN    Action: moveLeft    Reward: -0.35  NewState: NNNN
NWWN    Action: moveUp   Reward: -0.35  NewState: NWNN
NWNN    Action: moveDown    Reward: -0.35  NewState: NWWN
NWWN    Action: moveUp   Reward: -0.35  NewState: NWNN
NWNN    Action: moveDown    Reward: -0.35  NewState: NWWN
NNNN    Action: moveRight    Reward: -0.35  NewState: NWNN
NNWN    Action: moveUp   Reward: -0.35  NewState: NNNN
NNWT    Action: moveRight    Reward: -0.35  NewState: NNWN
NNNN    Action: moveDown    Reward: -0.35  NewState: NNWT
NNTW    Action: moveRight    Reward: -0.35  NewState: NNNN
NNTW    Action: moveDown    Reward: 10  NewState: NNTW
WNNW    Action: moveDown    Reward: -0.35  NewState: NNTW
```

Live demo:
https://ramtinkermani.github.io/MachineLearningCapstone/

Github page:
https://github.com/ramtinkermani/MachineLearningCapstone

Demo video:
https://www.youtube.com/watch?v=Y-Iz429xREw

Problem Statement

In reality it’s not always feasible to let the robot to run 24/7 for weeks just to learn about the house and simulation might be a better option for the following reasons:

1- Mechanical machinery have a limited lifetime

2- It will take a long time for the agent to be trained

3- When dealing with real world scenarios we are talking about continuous space which presents its own issues that we will talk about later.

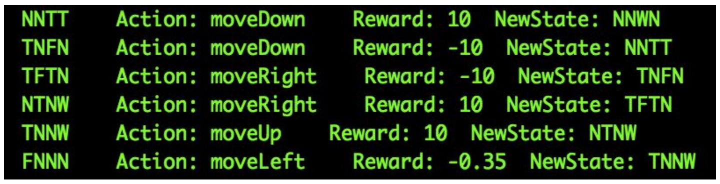
That’s where simulation and simulated environments and agents come into play. Simulated environments are not always the most accurate as they don’t represent the environment and physical world one hundred percent, however the advantages they offer makes up for the small inaccuracies.

Once the agent is trained in a sufficiently similar environment to the real world equivalent, we can export the model into a real robotic agent and with minor fine tuning, achieve a great result.

Metrics

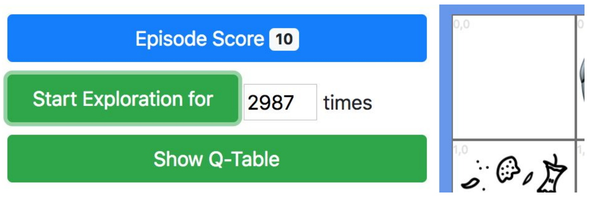
In Reinforcement Learning problems, two of the main metrics are the **immediate rewards** and **total score for each episode**.

In this application the user can observe the immediate score awarded to the agent due to taking a specific **Action** in a specific **State** in the Console window. As depicted in the following picture, as an example, the agent has received a score of 10 for “moving Down” when it was in State “NNTT” (**N**othing-above, **N**othing-left, **T**rash-below and **T**rash-Left) because it “Collected Trash”, a desired outcome. And has received a score of (-10) for “Moving Down” when in state “TNFN” (**T**rash, **N**othing, **F**ire, **N**othing: above, right, below and left cells, respectively) because it entered in to a Fire cell (an undesired outcome)



```
NNTT    Action: moveDown    Reward: 10  NewState: NNWN
TNFN    Action: moveDown    Reward: -10  NewState: NNTT
TFTN    Action: moveRight    Reward: -10  NewState: TNFN
NTNW    Action: moveRight    Reward: 10  NewState: TFTN
TNNW    Action: moveUp    Reward: 10  NewState: NTNW
FNNN    Action: moveLeft    Reward: -0.35  NewState: TNNW
```

On the other hand, what matters the most in either **Exploration** or **Exploitation** phases, is the **Episode Metric**. This score indicates how well the agent has performed completing the task and it’s shows with an integer value at the top of the left panel.



At the Exploration phase, the agent keeps moving around randomly and it’s learning how to behave in the environment to receive the maximum reward and the least amount of punishment. At this stage, since there’s no prior knowledge is involved and the agent is acting “in a dumb way” (like a baby trying to learn to crawl) and it’s not using the learned knowledge, the score if always very low, around 10-20.

However when the agent finishes the training phase and is ready to utilize it’s learned knowledge, we see that in most case it achieves a score of about 40 (which is almost the maximum score possible, when there are 4 trash cells in the environment, 10 per trash).

The episode score, shows clearly that the agent has indeed learned something. Also it’s valuable to note that a better and more realistic approach for Reinforcement Learning is to start utilizing the “Learned Knowledge” very early on so that the agent can learn while also occasionally using it’s knowledge. Initially most of the actions are random but as time passes by and the Q-Table becomes more valuable and populated, the agent uses that to perform and a tiny fraction of time, performs randomly to keep sploring new and possibly better options.

II. Analysis

Data Exploration

Inputs to this application are the environment specification which includes for simplicity the width and height of the environment and number of grids. Another set of parameters are location of dirt cells as well as hazard cells.

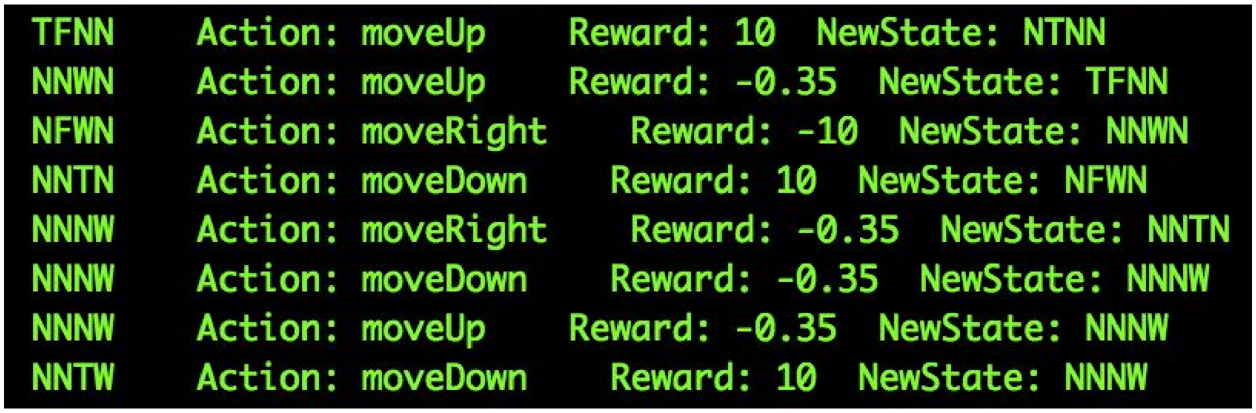
In Reinforcement Learning, the agent initially has minimum data about the environment, itself and its “**Purpose**”. It starts by running and performing tasks more or less randomly. These random actions puts the agent in various states and based on the action taken in each step, is awarded rewards or punishments. These rewards or punishments, become the learning states and for later steps, the data to the agent. Also the data based on which the agent is rewarded or punished could be considered the initial data of this learning algorithm.

The environment is a Web-based application and I used Javascript (Vanilla Javascript and JQuery) to create a simplified simulated environment that the robot will reside in, play around and learn its objectives and purpose.

The environment is a configurable grid of size (W square units x H square units). The number of **hazardous cells** (indicated with a fire icon) as well as the number of **dirt cells** (depicted with a trash icon) are configurable. However in every run, the positions of these cells are randomly chosen and placed.

Exploratory Visualization

The Console provides information about the immediate rewards given to the agent by the environment based on the action taken in each state. The total score is also displayed at the top of the left panel.



```
TFNN    Action: moveUp    Reward: 10   NewState: NTNN
NNWN    Action: moveUp    Reward: -0.35   NewState: TFNN
NFWN    Action: moveRight    Reward: -10   NewState: NNWN
NNTN    Action: moveDown    Reward: 10   NewState: NFWN
NNNW    Action: moveRight    Reward: -0.35   NewState: NNTN
NNNW    Action: moveDown    Reward: -0.35   NewState: NNNW
NNNW    Action: moveUp    Reward: -0.35   NewState: NNNW
NNTW    Action: moveDown    Reward: 10   NewState: NNNW
```

figure) Immediate reward shown on the console

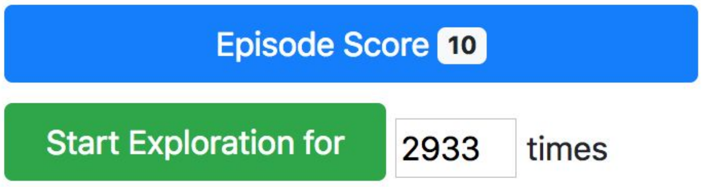


figure) Total episode score

Algorithms and Techniques

One of the most popular techniques in reinforcement learning is **Q Learning**[2] that is used to teach the agent learn about its purpose and goals and to perform efficiently. Initially, the agent does not know the rules and the fact that it’s a vacuum cleaner and its goal is to collect all the dirts and avoid the hazardous cells. However, as it starts moving around and performing actions, it will learn to pick the most efficient action in each step. This is called the agent’s **“Optimal Policy”**. These policies are stored in a table called the “**Q-Table**” which basically is a guide for the agent.

Each row in the Q-Table indicates a state and each column is one of the actions that agent can take in each state. The value of each cell is the “**Immediate Score**” the agent receives if it takes action A when in state X. The agent’s can perform in one of the following modes:

- **Exploration**: Agent is learning. It takes many random actions to “fill” in the Q-Table and explore the opportunities. To learn “What happens if I take action A when I’m in state X”.
- **Exploitation**: The agent has explored various scenarios enough that now has a relatively useful Q-table that it can exploit to actually perform the desired tasks. When in state X, it looks in the table and finds the action that gives the highest possible score and takes that action.

Benchmark

For this problem, the benchmark is the agent going around randomly in the environment and collecting dirt cells. This is the agent’s behavior in Exploration phase. Funnily enough this is almost the way most vacuum cleaners perform, well not quite randomly but not as intelligently as they could. Depending on the complexity and size of the environment, number of dirt cells and hazard cells, this benchmark will have different success rates, but in general, as we see, the success rate is very low, close to zero percent as the value of Trash cells and Fire cells are the same for the agent that has no idea about the environment (yet!)

After say 1000 episodes of executions, we can evaluate the performance of the agent using the following formula:

Performance = Total score of all episodes / number of executions

We use this benchmark to compare with the final performance of the trained agent. As explained before, compared to this benchmark, the trained agent performs at least three times better.

The average calculated result for 3000 random iterations in the Exploration phase is about 15. As we will see, when the agent is fully trained the average score goes up to about 40, three times improvement.

III. Methodology

Data Preprocessing

For Reinforcement Learning problems, we are not using any raw data to solve the problem. Instead we propose a benchmark and use scores to measure the performance of the agent in the environment.

Implementation

Algorithms and techniques were implemented:

After building the grid environment and defining different type of cells, the main data structure created to solve the problem was a Q-Table which is defined as a nested JSON dictionary in Javascript.

In the exploration or training phase, in each episode the agent is run randomly and forced to take a random action in each state. Based on the defined reward values, the agent is awarded a positive or negative reward and this value is added to the relevant Q-Table entry for that State/Action pair.

In the exploration phase, the robot, the fire cells and the trash cells are placed on the grid randomly and upon start, at each step the agent looks at the Q-Table content and chooses the action that maximizes its reward for that state.

Complications with the original metrics

The original metrics and defined rewards, included a time factor which defined how quick the agent finished the task. The primitive nature of the algorithm used and Q-Learning in general, forced me to remove the time factor so the success or failure is now calculated based on the number of Trash cells or Fire cells that the agent has landed on. Also initially, a higher grid was defined, the size of which was reduced to increase the speed of training.

Complications with writing the code

There were no issues implementing the algorithm and building the data structure for the Q-Table. I simply used a nested array to hold the score for each State/Action pair. The code is pretty self-documented and main functions are commented properly.

Intermediate solutions,

One of the gradual improvements I made while developing this application was defining the rewards given to the agent. Initially the values were very close to each other (Fire:-2, Trash:+2 and each step:-1) which caused the agent to not be trained properly. Later after modifying the rewards, I realized that the rewards given to the agent for walking into Trash and Fire cells should be much higher than a simple step (any movement, even into an empty cell). Also since the main goal was not to walk into a Fire cell (as in many real-world scenarios, it means the end of robot's life!) I tremendously increased the negative reward given to the agent when walking into the Fire cells.

Another issue was walking into the walls. Initially I set the rewards of walking into the wall (and hence not moving) the same as walking into an empty cell. This introduced the problem of agent keeping to hit the wall, since it had little to no negative rewards. After giving the agent a higher negative (-3) rewards for hitting the wall, the agent learned to choose an empty cell over walking into a wall.

Refinement

Due to the memory and time limitations I had to decrease the size of the grid to be able to achieve a reasonable running time. Also different values for rewards were exercised to achieve a better performance.

IV. Results

Model Evaluation and Validation

After a 3000 episode exploration phase, running the agent in Exploitation mode achieves an acceptable result. Out of 10 runs, about 9 are finished successfully and one run fails due to getting stuck in the local minima. We offer a solution later to resolve this issue.

However, score-wise the trained agent does much better than the random agent (the benchmark) for about three times the score value.

As mentioned before, the simple nature of this algorithm, doesn't allow the agent to see far in the future (or learn far from past). As a result, if the grid is too big and there are not a lot of Fire/Trash cells (i.e the grid is very sparse, full of empty cells) the agent won't perform well and might get stuck in local minimas way more often.

In general, this solution works great if there's high number of Trash and Fire cells in the environment.

Justification

This application shows the possibility of using a simple simulated environment to use for planning and routing applications of real-world agents.

I created a Web-based application intentionally for the following reasons:

- It helped create an interactive UI
- Privacy: Users data stays in her machine
- Training could be distributed to many machines and on user's browser
- Possibility of using new Web Based ML frameworks such as TensorFlow.js[4] and and DeepLearn.js
- No installation/ setup required
- Access to machine's sensors through the browser
- Possibility of accessing the demo via mobile phone, training and playing on the mobile browser

Final results compared to the benchmark result:

I exploration phase, after running the training for about 3000 episodes and taking the average of episode scores, we get an average score of 15. When the training is complete and agent is fully trained (in this case, the Q-Table is sufficiently populated for all State/Action pairs), after running the game (exploitation) for about 20 times, the average score is 40 which is more than 3 times improvement.

V. Conclusion

Free-Form Visualization

Running the application in both exploration and exploitation mode, shows metrics to evaluates the performance for both immediate and long term goals.

I have created a video, that shows the execution of the application in both Exploration and Exploitation phases. You can find the video at the following address:

https://www.youtube.com/watch?v=Y-lz429xREw

The exploration is in high speed to we can train the agent quickly, however in the Exploitation phase, the agent moves in 1-second intervals so that the user an follow its movements and observe its performance.

Reflection

The most challenging part of the project was to define the State space. I needed to define the "State". Problem is if the State contains too many variables, the state space will be huge and it will take a long time for the agent to learn, also the Q-Table become huge and we may run into memory issues. Initially I thought about defining the state as the following tuple:

Even if the grid is of size 4x4, each cell can either of following four states:

1- Hazardous cell   2- Dirt Cell   3- Robot   4- empty cell

That will add up to 4 to the power of 16 which equals to 4294967296. This means we need to fill 4294967296 rows of the Q-Table for each move (Up, Down, Right, Left) which is going to take a very long time and probably we will run out of memory.

I overcame this issue by discretizing the environment and making it small enough to produce a rather acceptable Q-Table size and acceptable performance and training time.

Description of the process going through implementation, refinement and result

The initial idea of this project was based on the inefficient movement of my iRobot Roomba in the house. I realized it could be beneficial if we can simulate the environment in which the robot is to perform and implement the learned policy on the physical robot performing in the real-world and hopefully after some tuning getting a better result. Initially, the application was too ambitious and I was thinking of continuous values for the robot movement and positions and a free form grid that resembles a house floor plan. Later due to the limiration (training time, memory in browser) I had to simplify the environment.

The first phase was the creation of the initial game environment which a user could play using the arrow keys (which is still in place). The next step was to add random movements and scoring to see how bad a random agent acts. And finally the third step was to implement the Q-Learning algorithm to keep track of agent's past experiences and actually utilize what it has learned in the future steps.

As mentioned before, after the initial implementation, steps were taken to improve the results. One was to simplify the environment and the other was tuning the parameters which in this case were the rewards given to the agent for it's behavior in the environment.

Improvement

The following features could be implemented to improve this application:

- I tied to vary the rewards awarded to the agent many times to improve the performance, however this still is not optimal. We might be able to use a method to tune the hyper parameters to achieve better performance and speed, both for training and exploitation.
- The inputs provided to the agent are somewhat granular. The environment provides the location of the robot itself and the Fire/Trash cells as X-Y coordinates of the cells within the grid of 4x4 hence a discrete state space. For a more realistic simulation, we can have the Image of the canvas to the robot (pixels) and give the robot the freedom to move freely in various directions and with arbitrary values. This will increase the size of the Q-Table tremendously and might take too long to train and we may run into memory issues. In this case using **Deep Neural Networks** for Reinforcement Learning (Such as **DQN** [3]) may be a better choice. This is the method used by **DeepMind** to train the computer to play Atari games (**Deep Q-Networks**[3])
- The grid could be expanded to be customized in different shapes instead of a square shaped grid to match real-world environments.

The following issues need to be addressed:

- The agent might get stuck in local minima as it pretty much values immediate rewards than the episode score to learn.
- Even though this application runs in a browser and I initially had the suspicion that it will run into memory issues, it didn't. However increasing the dimensions of States (currently 4) will substantially increase the size of the Q-Table and at that point I might need to consider other approaches such as **Deep Q-Networks**.

Regarding the application itself, the following features could be added:

- User should be able to save and reuse the Q-Table after closing the tab and even share a trained Q-Table for future use.
- User should be able to re-run the training to improve the learning farther. For example she should be able to run the training for 3000 episodes initially, and if the results were not satisfying or some State/Action pairs were not "experienced" by the agent (where the value of State/Action is the default 0), re-run the training for another 2000 episodes to improve the results.
- The environment parameters such as the reward values, grid size, number of Fire and Trash cells are currently hard-coded in the application (even though one can easily modify those values). We may expose these values in the UI.

- [1] http://media.irobot.com/2017-07-25-iRobot-Reports-Strong-Second-Quarter-Financial-Results
- [2] https://en.wikipedia.org/wiki/Q-learning
- [3] https://deepmind.com/research/dqn/
- [4] https://js.tensorflow.org/