WiDS Project Report

Project UID : 63

- Aryaman Prasad

Here is my report and various things that I found and learnt during the course of the project “How Machines Learn: A Journey Through Reinforcement Learning”.

(Note: All the problem statements spoken about in this report, except the final project, are from the book “Reinforcement Learning – Sutton and Barto” which was given to us as a resource for this project)

Week 1:

During this week we were given a ‘Multi-Bandit Problem’ which was sort of like a luck-based game, where we pull one among ten levers and get some reward. Despite seeming mostly luck-based, the algorithms we were asked to try and implement actually gave results with better average rewards compared to simply pulling levers at random.

The algorithm I implemented was quite similar to how a human would intuitively try to play the game, that is, if he finds a particular lever that gives him more rewards, he would majorly pull that lever only, while occasionally trying out some other levers as well in hopes of finding better rewards.

By changing the probability of exploring a random lever rather than going for the best available one, I got different average rewards, however the optimal probability was not a consistent value. In all likelihood, this was because the game is essentially still luck-based, and so, I could at best get a little range of optimal probabilities by running the code multiple times.

We were also asked to make slight changes to the algorithm and see how it affects the average reward, but overall, I felt each algorithm gave roughly the same average reward. Sometimes one particular algorithm would give a slightly better reward and sometimes it wouldn’t (Again, this is probably due to the fact that this game is essentially luck-based). All in all, every algorithm we applied was definitely better than simply pulling levers at random (which is probably the equivalent of a caveman-like person playing this game), which I believe shows that the machine is indeed learning something and not behaving like a caveman.

Week 2:

In this week, we were asked to formulate different situations as MDP's (Markov-decision processes). Essentially, from what I could tell, we have a situation where there are multiple states of being (for instance, a person is walking on an **N** x **N** grid, so each possible position of his is a state, making $\mathbf{N^2}$ states in total). Among these states, there are some particular states which we wish to or do not wish to reach. In game-like terms, there are some particular states that get us to win or lose (and eventually end) the game called terminal states.

Here, we were only asked to write Python codes to create the states, so writing the codes itself was fairly straight forward.

Week 3:

Here, we were asked to do two problem statements:

Jack's Car Rental Problem:

To briefly summarise this problem, Jack has two car stalls ('stall' might not be the right word but whatever) to rent cars. Each stall can have 0 to 20 cars at any time. Everyday, he rents out some cars and receives back some cars and has the option of transporting at most 5 cars from one stall to the other overnight to ensure availability of cars for rent at each stall. Now, we must find an optimal policy as to how many cars should be transported for all possible scenarios of number of cars in each stall to ensure the highest chance of availability of cars for rent (The problem also had some more specifications such as the fact that the numbers of cars rented and received were poisson random variables with a predefined expected number and also the rates of rent and transportation).

A humanly intuitive way of looking at this (in my opinion) is that, if one stall has an abundance of cars (close to 20 cars) and the other has far lesser cars, then transport cars from the former to the latter and try to balance the numbers of cars in each stall. If both stalls have an abundance of cars, live happily, and if both stalls have very few number of cars, then try to balance out whatever little cars you have in each stall and then just pray to God for a good outcome.

Interestingly enough, this is essentially also what the machine decides as the optimal policy. The idea was to generate each possible permutation of the number of cars in each stall as a state and implement a Dynamic Programming algorithm over it. To do this, we assign values to each state which in a sense give us the worth of each state and a reward as the money earned on a particular day. I won't go into explaining the entire algorithm elaborately, but I'll give a few things that I did or found in the process:

- For determining the optimal action from a particular state, I had to weigh the values of all possible next states and their consequent rewards. For my

program, I scaled down the rates of rent and transportation by 10 as doing that gave me a fairly satisfactory result.

- Perhaps one could weigh the values and rewards in different ways to get different policies. So to determine which is optimal, I tried accounting for the probability of the action ensuring availability of cars for rent. However, I did not actually iterate through various weights of values and rewards (like how I did for the Week 1 assignment) so my policy might in fact be sub-optimal.
- To determine the reward on a particular day, I could not simply generate a poisson random variable as that would lead to very high inconsistencies and ultimately give a seemingly random policy. So I instead generated 200 poisson random variables for each of our variables and took their average. Changing this number (200) to something higher could give more accurate results, or perhaps simply taking the expected number and considering the expected reward only (I actually realised I could do this much after writing the code with the previous approach and by that time, I was just too lazy to change it, I should probably apologise for that).

Gambler's Problem:

Essentially, there is a Gambler with some amount of money (ranging from 0 to 100), out of whatever he has, he can put some amount at stake and either double that amount or lose it completely. Reaching 100 is a win and 0 is a loss. Now, our job was to find an optimal amount to bet at any possible amount from 1 to 99.

We were asked to create an MDP for this and then generate an optimal policy and that was essentially what I did. Again, I won't fully elaborate on the algorithm itself but will just say a few things I did or found:

- If the probability of winning and losing a particular bet is 0.5 each, then the optimal policy comes out to be betting exactly 1 at nearly every instance (I did try this with my code, but what I shared would not have this as I had changed the probability back to what we were asked based on the problem statement). Essentially, the machine believes that it should try and play a relatively safe game and try to slowly increase its amount to 100. In fact, the optimal policy stays the exact same even when the probability of winning a bet is higher than losing.

- However, if the probability of winning a bet is lower than 0.5 (the problem statement gave it as 0.4), then obviously trying to bet exactly 1 every time wouldn't be a very efficient way, so there are some instances where the machine decides to bet higher amounts in hopes of increasing its amount massively in one go. In fact, this approach is sometimes slightly better even though it has a higher risk.

- This problem seemed relatively simpler to me compared to the Car Rental one, so here I only accounted for the reward that any state gave, which was 1 for an amount of 100 and 0 for any other amount. This was identical to the approach given in the book as well. Now, I had tried some minor changes, one notable change was setting the reward for an amount of 0 as -1, which I felt was completely valid to do. However, for certain probabilities, the code never gave an output and went into a seemingly infinite loop. I found this very odd and could really not come up with a plausible explanation.

Final Project:

The final project was about creating an algorithm using Reinforcement Learning to solve the 15-puzzle (it is a puzzle with a 4 x 4 grid with 15 tiles numbered 1 to 15 and an empty cell, with the aim being to shift the tiles around and get them in a specific order which we call the solved state).

The idea was not to have the machine generate the fastest possible solution, but rather to generate a more intuitive solution, similar to what a human would do if he were solving the puzzle. For instance, regardless of the initial position, a human would most likely try to first solve the first row, then solve the second row without touching the first, and so on. Now the fastest possible solution from the same initial position may not follow this logic and instead might consist of some moves which we as humans will find absurd and extremely unintuitive.

Before actually writing an algorithm, I had a few ideas of my own as well, so here is my journey through writing the code for this final project:

- Firstly, I had to generate the various states, obviously I did not simply consider all 16! states, and instead, depending on the number of rows solved in that state, I only considered the elements relevant to solve the very next row while the rest could be considered as all identical (something like 0). This idea was essentially what was given in a medium that was given to us as a reference for this project and it helped reduce the number of states very very drastically.

- Unfortunately the issue didn't end there, the total number of states was still somewhere around 6 lakhs and in order to do the value iteration and policy iteration, I needed a way to iterate through all the states. My first idea was to create a one-one map from each state to a natural number (since I know the total number of states). I tried doing this by ordering all the states in a lexicographic order and was able to create one-one functions from the set of states to natural numbers and vice versa. Although, implementing this ended up being quite inefficient so I had to scrap the idea.

- My other idea was recursively generating all possible states from a terminal states and storing them in some way, but this too was not very efficient.
- In the end, I looked at the java code given in the medium and took inspiration from it. Essentially, their approach to generating all possible states was similar to a recursive approach, except they didn't explicitly recurse through a function, but rather iterated through a list which initially consisted only of the terminal state but got all possible next states from there appended and the state being iterated removed. This was comparatively much more efficient.
- The value and policy iteration was mostly by the book (in terms of logic), however one little issue I noticed was that I had made 3 terminal states (1 row solved, 2 rows solved and full puzzle solved) so sometimes, the code might solve 1 or 2 rows and then either stop or loop back by disturbing a previously solved row. Theoretically, I'm not too sure why this happens, but I was able to correct it fairly easily by only allowing moves which increase the number of rows solved or keep it the same.
- Overall, I think the toughest challenge here was optimizing the code and getting it to work faster. My initial codes took absurdly long times to run (more than an hour or so) because of which I never got to see their output. Even after trying out multiple things and taking help from the medium, I could only reduce the runtime to about 4-5 minutes on my PC (it might still be somewhat high but for me it is a great success considering how long it was taking initially).

There are a few other things that I learnt or discovered while doing this project. Some of these things are incredibly small and (possibly) not very relevant to the project itself:

- The key for a Python dictionary cannot be a list, you get an error saying "unhashable type : List" which I find very awkward as I really cannot see a reason for Python to not allow a list as a key to a dictionary (I had to tweak my code a little due to this).

- When I was looking online for ways to optimize my code, many times, I came across the concept of hashing. Honestly speaking, I don't know what it really is or whether it can even be implemented in this algorithm, but I think it is something that can optimize codes and is something I'll probably learn in the future.
- Using classes to write the code made it much more well structured and easy to debug (I say this because I initially wasn't really using them).

Final Thoughts:

Over the course of this project, I got to learn the basics of a new machine learning algorithm that I knew nothing about before. From what I can tell, Reinforcement Learning is a form of machine learning that makes a machine think in a way very similar to how a human thinks. All the algorithms I wrote gave policies which were mostly intuitive and despite the policies being sub-optimal (compared to the fastest possible solutions), I find it very fascinating that we can get a machine to think somewhat like a human.

Since it resembles a human's intuitive thinking process, as the machine does the same thing again and again, it gets better at it. I think it is somewhat similar to how AI chatbots work nowadays (although I am only making a guess). Personally, I like to imagine a chess bot that knows the basic rules of chess and as it plays multiple opponents, it gets better at the game, but perhaps creating such a thing is too far-fetched for me (right now at least it probably is).

Anyways, what I'm trying to say is that perhaps there are ways to train a machine and make it think like a human, which I feel is something very extraordinary.