PROJECT: INVESTIGATING REINFORCEMENT LEARNING

SIT215 Artificial and Computational Intelligence

GEORGA LEISEMANN

SN: 218044006

Reinforcement learning is “is the training of machine learning models to make a sequence of decisions”. (Osinski 2018, part 1) The goal is to teach an agent to make decisions that will result in the optimum solution for the problem. This report will use environments provided by OpenAI Gym to examine reinforcement learning in different scenarios.

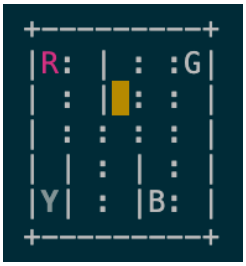
The Taxi Problem

The Problem

The Taxi Problem involves creating a program which finds the best (shortest) path for a Taxi (agent) to take in order to pick up and drop off *n* number of passengers. The problem is set up on a 5*5 grid with each square in the grid indicating a possible location for the agent.

There are also 4 squares within the grid which indicate pick up and drop off spots for customers. These are labelled R, G, Y, B and with pink indicating the pickup square and dark grey indicating the drop off square. The agent is represented as a yellow rectangle.

There are also 6 short lines. These represent walls that the agent must got around in order to pass. An example initial set up for the environment can be seen below:



In order to solve this, the problem needs to be broken up into possible actions, states and rewards as defined by the Markov Decision Process (Aflak 2018, part 2). The Action Space represents what the movement of the agent will be and is defined follows:

1. South
2. North
3. East
4. West
5. Pickup
6. Dropoff

The State Space represents all of the possible situations the agent can be in. The total number of situations is equal to 500 as there are 4 destinations (4*5), and 5 passenger locations (5*5). The calculation is therefore as follows:

$$destinations * locations = len(State\ Space)$$
$$(4 * 5) * (5 * 5) = 500$$

The environment will initialise a state and the agent will attempt to make the most optimal moves in order to pick up and drop off the customer while taking the shortest path. Rewards and penalties will be awarded in order to train the agent to make optimal moves and thus successfully complete their task (complete an episode). The rewards and penalties are stored in a reward table and work as follows:

- A high positive reward is awarded for a successful drop off (20)
- A penalty is awarded if the passenger is dropped off in the wrong location (-10)
- A slight negative reward is awarded for each action in which the passenger is not successfully dropped off or is picked up. (-1)

Random Policy

One way to solve this problem is to implement a Random Policy algorithm. This involves the agent making random moves until they correctly pick up and drop off a customer using the reward table. The reward table contains the reward for each state with the highest reward (20) being equal to the agent's destination. The program can therefore randomly search the space until the destination is met using the following steps:

```
While episode is not complete (reward != 20):
       Agent takes a random action
       Update current state to the new state
```

Using this method over 100 episodes, the following metrics can be obtained:

```
Average timesteps per episode: 2523.92
Average penalties per episode: 817.03
```

It is evident that this is a highly ineffective method as there are a large number of timesteps before an episode is complete. Other methods need to be explored in order to optimise the program.

Q-learning Algorithm

The Q-learning algorithm uses the environment's rewards to produce a Q-value which represents how beneficial each action is and thus learn the optimal route for the agent to take. The Q-value maps to a combination of state and action and therefore if the agent is in a certain state, they can look up the Q-value to determine the best action to take (i.e. the highest Q-value). (Kansal 2020, part 11)

An example of this is if an agent has picked up a passenger and has now arrived at the correct destination for drop off. In this situation, the Q-value for the 'drop off' action would be much higher than any other action.

The mathematical model that defines Q-value is as follows:

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\left(reward + \gamma \max_{a} Q(next\ state, all\ actions)\right)$$

(Kansal 2020, part 11)

In this equation, alpha is the learning rate ($0 < \alpha \leq 10 < \alpha \leq 1$) which changes the degree to which the Q-values are updated on every iteration. Gamma ($0 \leq \gamma \leq 10 \leq \gamma \leq 1$) is the discount factor which, when high (close to 1) captures the long-term reward and when low (close to 0) captures the short term reward (i.e. reward for the next action rather than all of the actions involved in an optimal solution).

The arrow indicates that the right side of the equation is being assigned to the left side (i.e. the Q-value of the agent's current state). On the right side of the equation, the weight of the old Q-value is added to the learned value. The learned value is calculated by combining the reward for the current state's current action with the discounted maximum reward for the next state. (UNSW 2020, part 1)

Once a Q-value is calculated for an action in a certain state, it is added to a Q-table which is matrix of states * actions with the Q-values in each cell. Its structure for the Taxi problem can be seen below:

		Actions					
		South(0)	North(1)	East(2)	West(3)	Pickup(4)	Drop off(5)
States	0	0	0	0	0	0	0
	1	-					
	2						
	…	…	…	…	…	…	…
	499						

One final element is necessary in order for the Q-learning algorithm to work effectively. A parameter called epsilon (ϵ) is used to ensure that the algorithm uses both exploration to find Q-values and exploitation to choose the optimal action. If this extra parameter were not used, the algorithm may continuously take the same path and thus may overfit. It works by comparing the epsilon value to a randomly generated value between 0 and 1 and making a decision based on whether the epsilon value is larger or smaller.

The program implements the Q-learning algorithm to train the agent using the following steps:

```
For 10,000 episodes:
      While episode is not complete:
            Compare Epsilon value to randomly generated value
            If Epsilon value is greater than random value:
                  Explore the action space
            Else:
                  Exploit the learned values
            Update q table with new Q-value
            Update current state to the new state
```

Once this method has been implemented, the training is complete, and the Q-table has been filled in. The following steps can then be used to solve the problem using Q-learning:

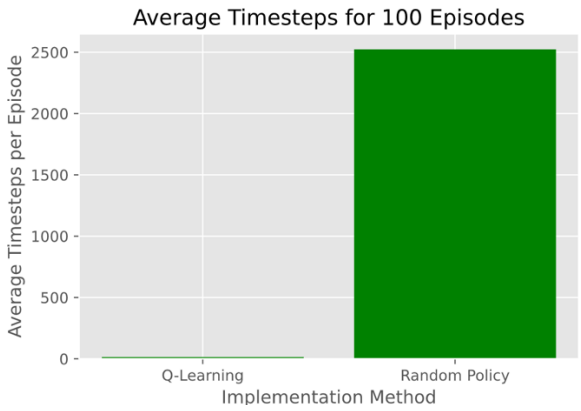
```
While episode is not complete:
      Agent takes action based on best value in Q-table
      Update current state to the new state
```

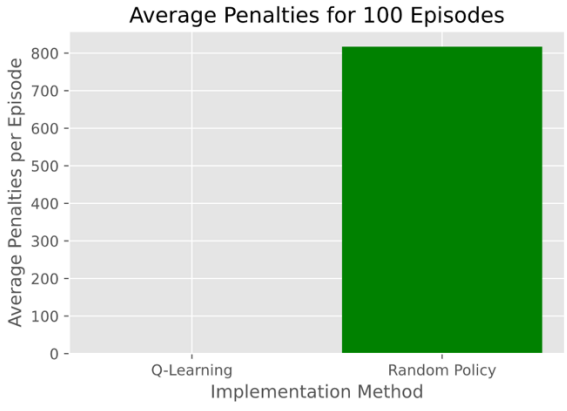
Using this method over 100 episodes, the following metrics can be obtained:

```
Average timesteps per episode: 13.17
Average penalties per episode: 0.0
```

Q-learning VS Random Policy

When comparing the metrics for Random Policy and Q-learning, it is evident that Q-learning is a far more efficient algorithm for this problem. This can be seen in the graphs on the following page:





It is clear that the Random Policy algorithm makes thousands of extra steps and incurs hundreds of extra penalties than the Q-learning Algorithm. This is due to the fact that the Random Policy agent is not learning from past mistakes. As the actions are random, the same actions can be taken over and over without the agent ever learning that this is not an optimal action.

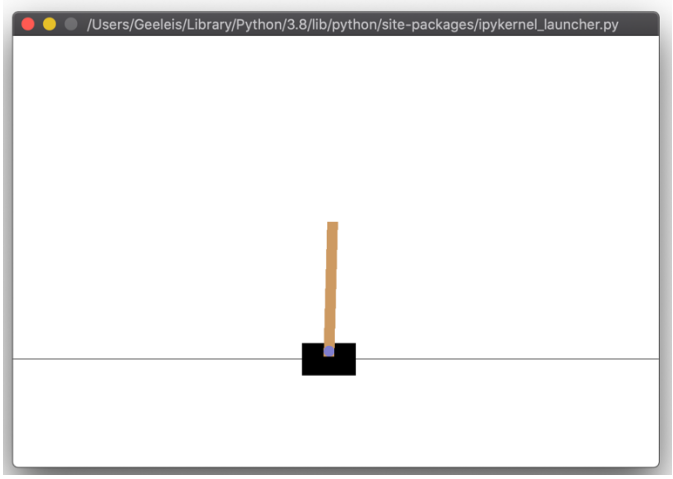
When using reinforcement learning, such as Q-learning, this repetition of mistakes is not possible as the agent has already learnt which actions are sub-optimal and thus will likely not take these actions again.

The CartPole Problem

The Problem

The CartPole problem involves moving a cart (the agent) either left (-1) or right (+1) in order to keep a pole which is balancing on top of the agent from falling over. The movement by the agent is classified as the action.

This problem is very similar to the Taxi problem in that it utilises the Markov Decision Process by breaking up the problem into possible actions, states and rewards. The visualisation for this environment is as follows:



The environment provides several float variables which define the state. These include the following:

- "The position of the cart
- Its velocity
- The angle of the pole
- The velocity at the tip of the pole"

(Maynez 2020, part 2)

The reward for this problem is modelled as +1 for any timestep in which the pole does not fall over. The overall reward is a sum of each of the rewards awarded during the episode.

Random Policy and Q-learning have been used in an attempt to maximise the number of timesteps that occur within each episode (i.e. maximise the time before the pole falls over).

Random Policy

For this problem, Random Policy was implemented in using the same method that was used for the Taxi problem. That is, random moves were made with the hope that these moves would keep the pole from falling. Using this method over 1000 episodes, the following metrics can be obtained:

```
Average timesteps per episode: 22.122
```

Q-learning

To begin our implementation using Q-learning, it is necessary to discretise the variables for state as they are provided as continuous and thus the possible combinations for each state is extremely large. This would cause the training process to be very inefficient. In addition to this, some of the variables have an infinite range which need to be given upper and lower bounds. To discretise, a set of buckets have been created and the continuous variables have been sorted into these buckets as follows:

```
Create buckets with upper and lower bounds:
For variable in observed variables:
      Scale variable value to that of corresponding bucket
Digitize variables and sum to a single integer that represents state
```

Once discretization has been completed, the Q-table can be initialised. This table will take on a different shape than the Taxi Q-table as the number of states and possible actions differ. There are two possible actions, as mentioned before:

1. Move Left (-1)
2. Move Right (+1)

The number of possible states can be defined as follows:

$$len(state\ space) = len(max\ bucket)^{len(buckets)}$$

This mean that the Q-table will take the following form:

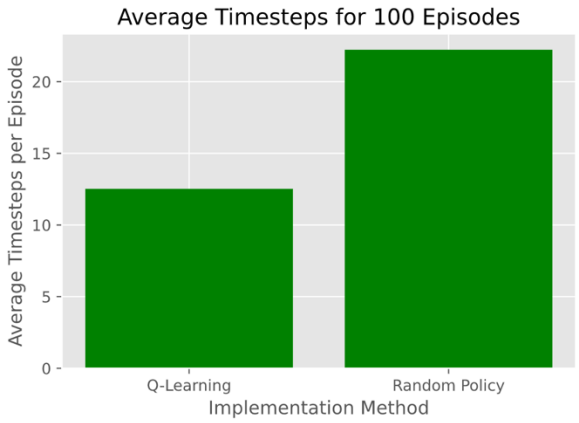
		Actions	
		Left(-1)	Right(1)
States	0	0	0
	1	-	
	2		
	…	…	…
	624		

Once the Q-table has been defined and the state has been discretised, the same process for Q-learning can be followed as what was used for the Taxi problem. Using this method over 100 episodes, the following metrics can be obtained:

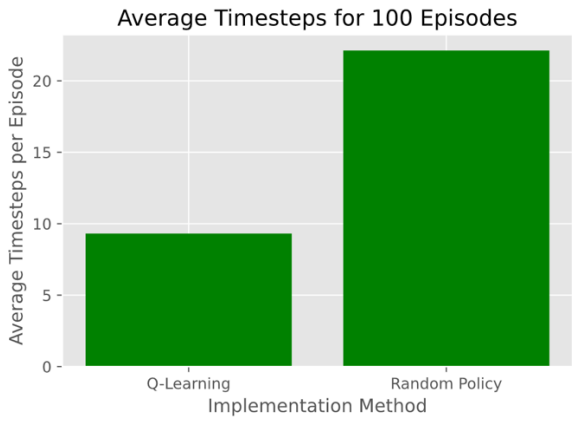
```
Average timesteps per episode: 12.52
```

Q-learning VS Random Policy

For this problem, our goal is to maximise the number of timesteps per episode rather than minimise them as per the Taxi problem. Therefore, when comparing metrics for Q-learning VS Random Policy it is evident that Random Policy is the more efficient algorithm. This can be seen below:



Test 1



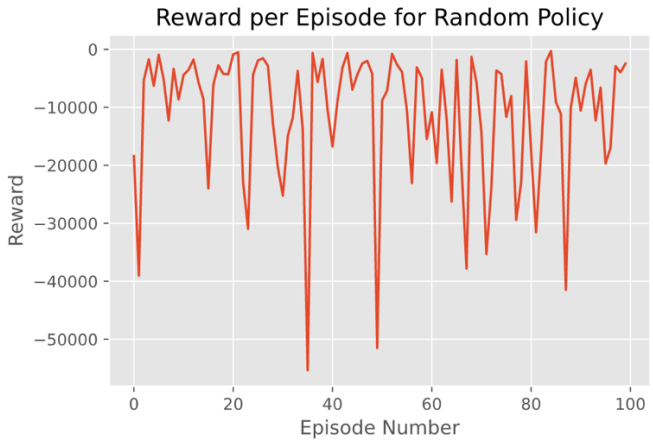
Test 2

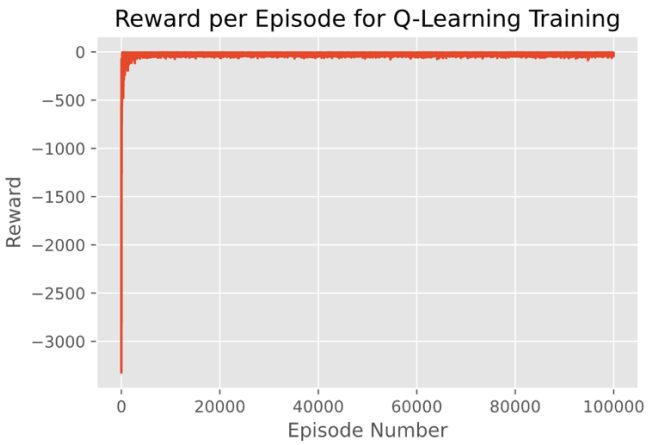
It is clear that the Q-learning algorithm has less timesteps than Random Policy in both tests (as well as all other tests executed previously), thus performing worse. The hyperparameters were tweaked and tested many times, yet the same results were still seen. The poor performance of the Q-learning algorithm will be discussed with respect to the Taxi Q-learning algorithm on the next page:

Q-learning Algorithm: Taxi VS CartPole

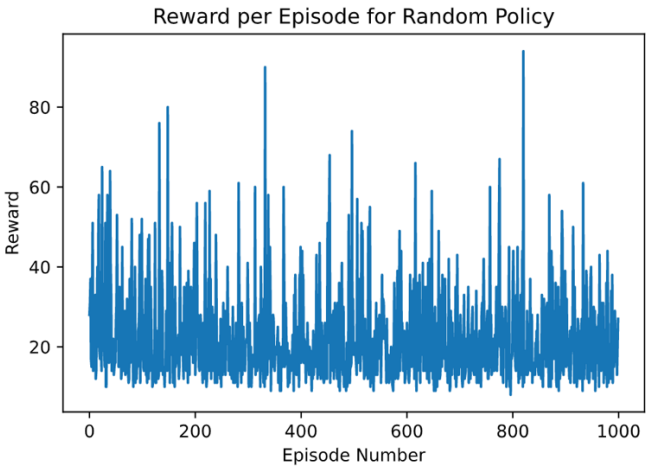
When looking at the performance of the Q-learning algorithm for both the Taxi problem and CartPole problem, it is evident that, despite the implementations being almost identical, the Q-learning algorithm for the Taxi problem performed far better than the Q-learning algorithm for the CartPole problem. This can be seen in the following visualisations which depict the average reward the agent receives per episode for both Q-learning training and Random Policy. As the value of rewards are different for both problems, the range of values also varies on the graphs. However, the main takeaway from these visualisations is the overall shape of the graph:

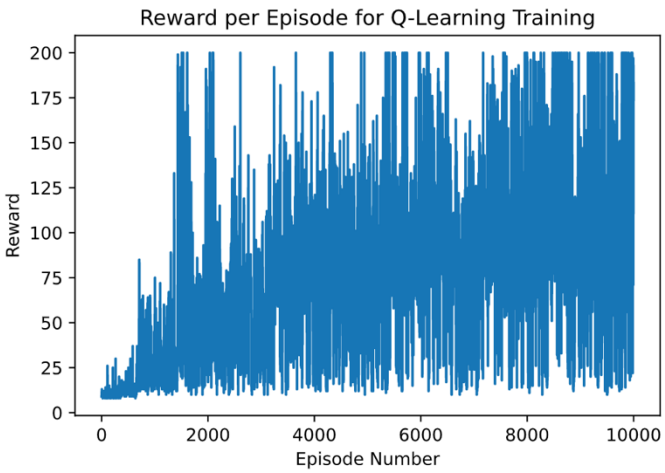
The Taxi Problem





The CartPole Problem





In the Taxi Problem graphs, it can be seen that when using Random Policy, the total reward jumps from being very low to close to zero, over and over. This is because the agent does not learn as it goes and thus repeats mistakes. In the Q-learning algorithm however, the reward per episode starts off very low and after a few thousand episodes it appears as though the agent has learnt the most efficient paths and therefore the reward per episode remains at approximately zero.

In the CartPole graphs however, it can be seen that the Random Policy algorithm causes the agent to act in a similar way, but the Q-learning algorithm does not appear to continue learning until an optimised point is reached. Instead, the agent appears to be learning until approximately the 1900th episode. At this point, it is evident that the agent repeats mistakes over and over, as with the Random Policy implementation.

This could be due to the fact that in some cases, the algorithm is greedy and acts in accordance with a short-term reward, rather than aiming for a temporarily more costly long-term reward. This causes some actions to be overweighted, and thus repeated.

We see this issue in the CartPole problem and not in the Taxi problem as the ratio of states to actions is much lower and thus the possibility of overfitting becomes much greater. In order for Q-learning to be effective for CartPole, the parameters would need to be tuned to high degree for optimisation, using substantial time and resources.

References

Aflak 2018, Math of Q-Learning with Python Code, Medium, retrieved 16 September 2020, <https://medium.com/datadriveninvestor/math-of-Q-learning-python-code-5dcbdc49b6f6>

CSE 2020, Reinforcement Learning, UNSW, retrieved 17 September 2020, <https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html>

Kansal 2020, Reinforcement Q-Learning from Scratch in Python with OpenAI Gym, Learn Data Science, retrieved 16 September 2020, <https://www.learndatasci.com/tutorials/reinforcement-Q-learning-scratch-python-openai-gym/>

Osinski 2018, What is Reinforcement Learning? The Complete Guide, Deep Sense AI, retrieved 18 September 2020, <https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/>