Self driven car with Q-Learning

Jacob Nyman
Viktor Sandberg

Abstract—AI and Machine Learning are one of the most popular subjects within computer science today. An area within Machine Learning is Reinforcement Learning where an agent learns good behavior through rewards and actions. In this report the Reinforcement Learning algorithm Q-learning have been used to produce a game where a car can learn to drive a race track by itself. This report covers how one way of achieving this goal can be done and the important realizations during the project. One of which is the importance of starting simple in order to achieve progress in a linear way. The result of the project was a car that was able to learn a large variety of tracks and the user could also race against the agent.

Index Terms—Q-learning, racing game

1 Introduction

Autonomous driving is a relatively new field in technology that has in the past 10 years gone from fiction to something inevitable. Many of the top companies have spent billions of dollars in research to be first on the market with the self driving cars. But how does this research work? How can you teach a car to drive itself? The answer to both these question is Machine Learning. Machine Learning is a method of data analysis that lets a computer explore different tasks without any human interaction, but instead relying on patters and statistics. In this project, the Machine Learning technique Reinforcement Learning and the algorithm Q-learning was used in order to achieve a simple model for a car to drive autonomous through a city while at the same time having an interactive game where a player is competing against the agent to reach the goal first.

2 Background and Related Work

The following chapter will explain the theory and the techniques used in this project and also how similar and related work became inspiration for the end product.

2.1 Reinforcement Learning

Reinforcement Learning is an area of Machine Learning where an agent seeks out cumulative rewards. Reinforcement Learning uses a concept called Markov Decision Process(MDP) in order to teach the agent about the environment. An MDP is an extension of the Markov chains but also includes an agent and a decision making process(actions). An MDP is defined by the steps presented in 1.

1. Set of possible States: $S = \{s_0, s_1, \ldots, s_m\}$
2. Initial State: s_0
3. Set of possible Actions: $A = \{a_0, a_1, \ldots, a_n\}$
4. Transition Model: $T(s, a, s')$
5. Reward Function: $R(s)$

Fig. 1. The different components of the Markov Decision Process

The reward function in Figure 1 $R(s)$ returns a value every time the agent moves from one state to another. With the help of the reward function it is possible to say what states are more desirable than others. This is because the agent get a positive reward from moving to that state. However, this can tell what states are undesirable as moving to that position will result in a negative reward. Then *how* do the agent maximize the reward and keep away from the states that result in a negative reward? The solution is to find a policy $\pi(s)$ that returns the action that leads to the highest reward. To do this, Q-learning is used because the goal of the algorithm is to find the actions that leads to the highest reward [2].

2.1.1 Q-learning

Q-learning is an off policy Reinforcement Learning algorithm. An off policy algorithm is an algorithm that learns from actions outside the given policy, i.e learns from taking randoms actions. When Q-learning is performed a *q-table* is created which is a matrix where each possible state is represented by a cell. In each cell all possible actions for that state is stored. The q-table is used by the algorithm in order to store and retrieve the so called *q-value*. The idea is that the agent will choose the best possible action in a specific state based on the q-value. The q-value is decided through Equation 1.

$$Q^{new}(s_t, a_t) \leftarrow (1-a) * Q(s_t, a_t) + a[r_t + \gamma * maxQ(s_{t+1}, a)] \quad (1)$$

- $Q(s_t, a_t)$ is the old q-value.
- a is the *learning rate*. The learning rate decides how much new information the agent should consider.
- r_t is the *reward*. The reward is different for any given state. But if the agent reaches the goal state the reward will have its max value to empathise that this is good behavior.
- γ is the *discount factor*. γ determines how much the agent should consider future rewards over current rewards.
- $maxQ(s_{t+1}, a)$ is an estimate of the optimal future value.

With the help of the Q-learning algorithm in Equation 1 it is possible to decide a policy which will return the actions which will lead to the highest reward. But in order for the algorithm to do so the agent has to learn. Q-learning have two approaches, either learning by exploration or by exploitation. When the agent is exploring it does random actions to be able to fill the q-table with as many new q-values as possible. When the agent have explored and filled the q-table it can start exploiting. An agent that is exploiting is using the q-values in the q-table rather then deciding new ones. By using a q-table filled with values the agent can decide what action in each state that will lead to the maximum reward and therefor becoming the policy of the Reinforcement Learning algorithm [4].

Jacob Nyman and Viktor Sandberg is with Linköping University, Sweden, e-mail: jacny980@student.liu.se and viksa378@student.liu.se

2.2 Related Work

An inspiration for this project was the Youtuber *Code Bullet*'s video *A.I. Learns to DRIVE* . In that video, the Youtuber explains the process of how he uses Q-learning to teach a car how to drive. By using ray casts and breadcrumbs he was able to get the car the optimal and fastest way around the course [1]. Other than Code Bullets video, the article *Q-Learning introduction and Q Table - Reinforcement Learning w/ Python Tutorial* by "Python Programmer", gave a deeper understanding of how Reinforcement Learning worked, and how it could be used in an application for a self driven car [3].

3 Method Implementation

To achieve an autonomous driven car simulation, the following method and implementation was used.

3.1 The Game Structure

The implementation process was made in Python with the usage of the game framework PyGame. To implement the Q-learning algorithm it was decided that the best approach would be if the game was created from scratch so that all behaviour of the application was known. A simple tile based car game was therefore implemented with the possibilities for a user to drive a car on a map with walls on which the car would collide. The car's movement was not bound to the grid sizes but instead the car had acceleration and turning based on for how long the turn button was held. Figure 2 shows the first iteration of how the game looked with the car in the top left corner, the walls painted red and the yellow square as the goal.

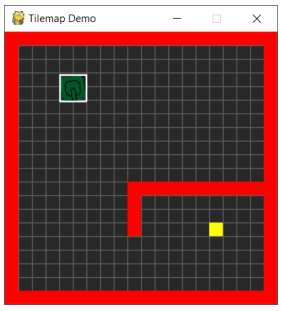

Fig. 2. The first look of the car game

The collision detection in the game was achieved with PyGame's sprite collision engine that every frame checks if the car sprite is within another and makes it possible to stop the car if that is the case. The purpose of the game was for the car to reach the yellow square, without hitting any walls. If a wall would be hit, the game restarts. It was thought that this was a simple approach that the Q-learning algorithm would have no problem to complete.

3.2 Q-learning and implementation

The Q-learning implementation consisted of many steps to reach the final result. This was due to inexperience which led to lots of experimentation. The different experiments will be discussed in the following sections.

3.2.1 Experiment I

After the game was created and the code structure decided, the Q-Learning algorithm was implemented into the game. The first implementation used a q-table that was 8 times the size of the actual grid size. In each of the cells of the q-table a q-value for each possible action was stored. The size of the q-table made it possible for the agent to have a unique key for every position relating to the goal position. That size was also chosen because it would make sure that the table would contain enough data for the car to be dynamic and know how to handle new situation if anything unexpected occurred.

When a q-table was chosen, the actual Q-Learning algorithm was implemented into the game. Every game loop makes the algorithm choose an action to make. The action was chosen depending on what the highest q-value for the current position in the q-table was. To retrieve a q-value from the q-table, the agent's position as well as the distance vector to the goal was used as the index in the table.

To fill up the q-table, and teach the agent how to drive the car was made to drive around the course without drawing the movement onto the screen. What was later noticed was that the agent even after thousand attempts didn't learn anything and that it only drove in circles. Another approach was needed.

3.2.2 Experiment II

The second approach was the realization that our first model was too difficult for the agent to learn. This was manly because the agent never moved far enough to reach a new q-value because of the acceleration, instead just going in circles. Even when the acceleration was set higher, the car learned that it got the best result from driving into the walls, dying. The agent never explored enough to reach the goal.

What was understood from this was that no matter how long the algorithm ran, the agent never learned because nothing motivated it enough to explore the map. It was then decided to implement *breadcrumbs* along the map to encourage the agent to explore more. These breadcrumbs were simple point gaining positions along the map, which would supposedly motivate the agent to take this path on the map. But even this approach didn't seem to work as expected: instead the agent took the first breadcrumbs, and then killed itself. When the punishment for dying was increased to avoid death, the agent went back to drive in circles again.

3.2.3 Experiment III

The third attempt of trying to get the agent to learn came when it was decided to go back to the very basics. What was done, was that a *simple tile to tile* movement with 4 different moves was implemented instead of the more advanced *acceleration, hold-to-turn* movement. When this was used instead, it was immediately seen that the agent made significant improvements when the the learning process was running on simple maps. Problems occurred later when the agent were exposed to more difficult environment. Since it was known that the model worked, the problem was that the agent needed more time to learn. But it took over 12 hours to run 25000 episodes, without any large improvements of the agent. Therefor an optimization of the implementation needed to be done in order for the Q-learning algorithm to run faster. The optimization consisted of changing the collision handling from going through all the sprites in the game to only check if the cars position is a wall position. This change made it possible to go from 25000 episodes in 12 hours to 25000 episodes in under a minute. It was now possible to set the agent in more difficult environments and quickly see improvements.

3.2.4 Final implementation

For the final implementation, the movement was increased to 8 different direction, instead of 4. The movement was chosen because it was the closest thing to a hold to turn movement, were the agent learned from exploring and that still looked close to a natural car movement.

The q-table was also changed to be 3 dimensional with the the the number of different actions q-value stored in each slot. The first dimension of the table was the rounded distance to the goal, and the 2 other dimensions were the position of the agent in the grid. Figure 3 shows a simplified version how one part of an iteration of the algorithm could look. On this position, the distance to the goal is 2 and the cars position is at (2,3). If this state or observation(obs) is put into the q-table, the hovering matrix are the values stored in that position of the table. Number 13 has the largest value, and is therefore chosen as the next action.

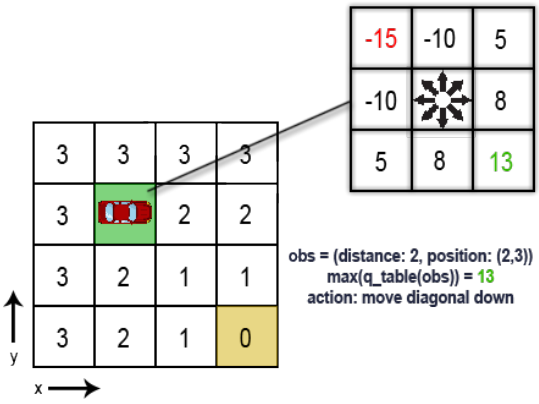

Fig. 3. Movement deciding

3.3 Interaction

The last thing that was implemented, was the possibility to compete against the agent by playing another car on the track, trying to reach the goal first. Both cars are placed on the track at the same time, and if a car dies or reaches the goal, the game restarts.

The challenge here was because the Q-learning algorithm needs to run continuously and at the same time the game also needs to listen to user input, threading was tried in order to be able to do two things at the same time. However, to have more control over the Q-learning algorithm and the game, instead of using threading, the Q-learning algorithm was instead called every other game loop. This way it was easier to get the same speed of both cars, and also to get a better playing experience.

3.4 Map Creation

The final map was made using the program *Tiled*. Tiled is a program that makes it easy to create tile maps with collisions and art and then export them for usage in Python. Using the tile set Gallet City created by the artist *Adam Saltsman* a map was created and then imported into PyGame.

4 Results

The result of this project was not only an interactive racing game, but a much deeper understanding about A.I and Machine Learning. When the final q-table and final model was created, 2 million tries or episodes, of training was run, and the result of that training can be seen in Figure 4. After the first million episodes, the table was tested, but decided to run 1 million episodes more for more extensive learning.

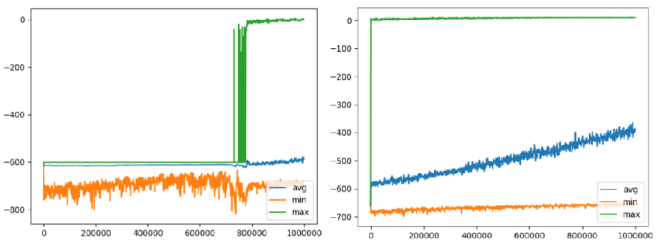

Fig. 4. The learning rate

One sample in the graph, correspond to 1000 episodes. The green curve shows the best result in one sample, the blue is the average value and the yellow is the worst run. The importance of this graph is the blue curve, the average, increases over time which is equivalent to that the agent is learning.

The final implementation can be seen in figure 5. Where the red car is the agent and the green car is the player.

Fig. 5. Playing against A.I

5 Discussion

The results of the project were not what was initially expected. What was expected was more grand than the one that was produced. The result was expected to be like the earlier mentioned inspiration video *A.I. Learns to DRIVE*, but even more advanced and with more features. This was a naive approach to the project as we did not have any prior experience with Q-learning and Machine Learning in general. As mentioned in the implementation chapter in subsection 3.2, there were many problems with getting the agent to learn. Even using ray casts to detect walls for the agent was tried, but the agent still did not seem to learn even as more and more complicated observations for the agent were added.

Because of our naive approach we thought the problem was because the agent did not learn for long enough or that there was a problem with the size of the q-table. But as both those parameters were changed and no improvement was made, we eventually saw that the problem was with the agent's movement. The agent started seeing improvements in learning as soon as the simplified movement had been implemented. It was here we had realised our naive approach and that we should have started the project with a much simpler model. The final movement is with 8 different moves that give the agent a car like appearance when it moves, but not exactly what we wanted to accomplish. To be able to accomplish the car-like movement with acceleration and a turning radius much more time and effort would be needed.

The problem with the first approach was that the agent did not get to the wanted states and did not seem to choose actions that led to a higher reward. The reasons behind this problem can be many, but one of the reasons that was realised was that the agent often were in the same states even after an action as the action did not take the agent to a new state but the same as before. Another reason for the problem with the first car-like movement is that the q-table did not represent good states for the agent. We thought of a solution to that problem by having the angle of the car and its position as the states that was stored in the q-table. Unfortunately there was not time left in the project to try this in practise.

Another part of the project that was problematic to implement was an interactive element. The first interactive element that was intended to be implemented was to be able to place obstacles on the course that the agent had to avoid. This element was intended to be dynamic so that the agent could avoid the obstacle wherever it was placed on

the track and to be able to avoid it every time. The problem with this approach was that it was possible to do, but in order to do it you needed a very large q-table that we developed, but it was impossible to run on a computer with mid-level-performance, because it took too much memory to read it. That q-table's size was 1.8GB. This approach was tried with smaller q-tables but they were not big enough so some states shared values between the different positions of the obstacles. The states that was observed in the obstacle implementation was euclidean distance to goal and difference in position to obstacle. What was later realised was that the agent could have been trained on the environment instead of the obstacles. The agent would then train on the different positions of the obstacles. By doing this the agent would not have had to learn all possible positions of the obstacles.

Something we realised midway through the project was the impact of small differences in variables and how much difference it really had on the result. The first realisation was about how important understanding the variables of the Q-learning algorithm was. In the beginning mostly different values were tried to get a good result, but the result did not improve. The improvement came when time was taken to understand them more deeply and by that started seeing some better results. One variable that took a long time to understand was *epsilon* that we realised was the most important one. Because the decay of that variable really impacted how much the agent learned.

Besides the problems throughout the project the result of the project was better then expected. All of the problems throughout the project have given us insight about Q-learning and Machine Learning that we otherwise would not have gotten.

In the result the agent always finds the optimal way to get the highest reward on all the tracks that were tried and it does it on quite few episodes as well. Usually around 100 000 for large q-tables of the size 32x32x8 or even larger.

6 Conclusions and Future Work

In this project we have stumbled on many problems and the road to the end result have not been the easiest, but the knowledge acquired on the way have been worth it. The most important insight from this project is to start simple, really simple and build upon that. The level of delicacy needed when working with Q-learning and Machine Learning is extensive and even the smallest change can have a large impact on the result. Something that was realised from this project was that you need to break your problem into small components, which in our case was the movement of the agent, that can then be developed into actions the agent can take. We have also learned about how many different ways it would had been possible to develop the Q-learning solution as you can use different kinds of states in your q-table, different actions and also other learning mentality with completely different variable values. The conclusion from this is that Q-learning is a very modular approach to Machine Learning.

To develop this project further we would try getting our initial car-like movement to work. This problem would be engaged by using the propositions made in the discussion chapter. Another thing that could had been continued working on would be the interactive part and especially the obstacle one.

Other future work would be creating a web-application for the program and try developing it in another programming language. The project could also have been further developed with neural networks in the form of *Deep Q-learning*.

This project have been a roller-coaster of emotions, successes and failures. But in broad terms this is much of what Q-learning and Machine Learning is. It is trial and error, sometimes you get a magically good result because you tweaked a variable a little and sometimes you do a change that seem like it will have a great impact but it did nothing at all. This is what we have felt and we feel like this has been a project that gave us much knowledge of the subject rather then producing an amazing result.

References

[1] C. Bullet. A.i. learns to drive.

[2] M. Patacchiola. Dissecting reinforcement learning-part.1, 12 2016.

[3] P. Programmer. q-learning-reinforcement-learning-python-tutorial.

[4] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.