Self Driving 2D Car

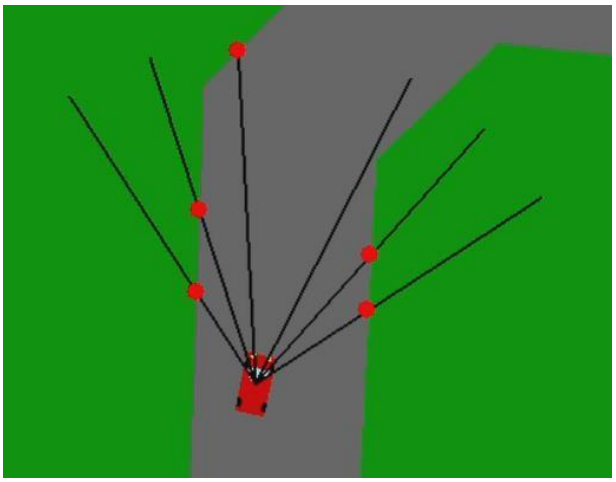
Introduction and Problem Statement

This project will use Deep Q learning, which is a type of trial and error deep learning method to predict optimal controls to beat a game or achieve a certain goal in a virtual environment. Many data driven deep learning methods are limited due to the amount of required pre-labeled data that needs to be provided. Situations such as the self-driving car often are impossible to be solved through data alone since the data on all possible ways a car can drive is massive.

Deep Q learning lifts such limitations, focusing on defining sets of rules through which the computer can explore a given problem on its own and collect data as well as learn from it with little to no human input. This is achieved through combining how humans learn (Q learning) with how human brain works (Dense Neural Networks).

Data Sets

Since my project was based on reinforced learning, the whole objective was to eliminate the hassle of data sets, so I have no datasets to provide, instead I will show how my car sees and collects data.



As you can see, the car mimics how 3D radio/sound/light waves work in terms of detecting nearby objects. In this case, it is a lot more responsive because we don't need the wave to bounce back to the sensor to tell how far the borders of the road are located. I initially had sensors pointing backwards too, but I found that backwards sensors would usually confuse the car and opted on using only front facing sensors.

As the car moves it is constantly pulling in distances between itself and nearby borders,

which help it “see” the track in its own way through numbers, which is what computers are great at analyzing. At each frame these 6 distances are passed onto the neural net, which feeds them to 2 hidden layers that output 2 values, 1 for left and 1 for right. Note that the car does not actually control the forward action, because often it would move extremely slowly to avoid dying. So, I decided to force the car to move forward constantly and have it learned how to control which direction it should go.
As the car moves forward, it receives good car points, but he loses all his points if he goes off the track. This system was to encourage it to keep learning until it was able to perfectly complete the track.

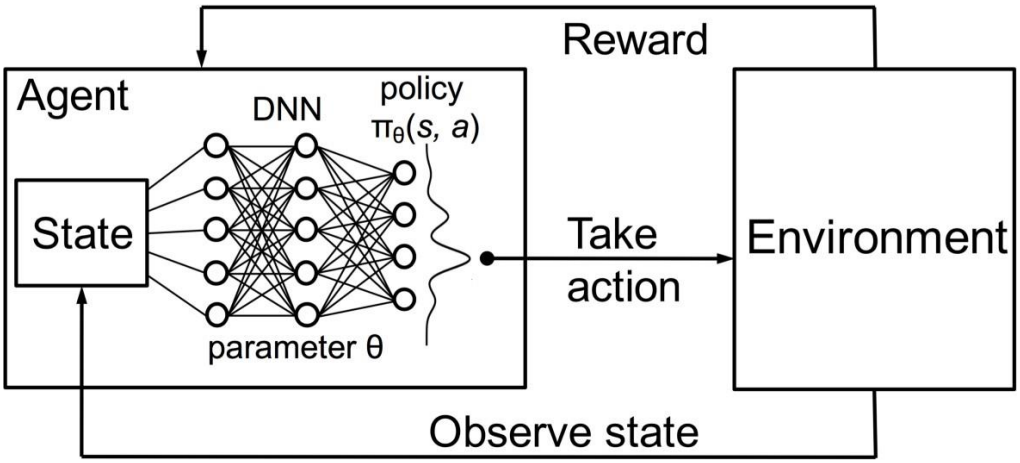
Description of Technical Approach

Initially I started out using openAI’s Gym library which had many environments for machine learning already created, however I quickly found out the amount of computation these games took, because almost all of them used frames captures as the main source of observation for the AI. 50 simulations took almost 30 minutes to run, thus I decided to explore alternative ways. This is how I found out about google using waves instead of pictures to teach the machine how to drive. I spent about a day trying to come up with a realistic plan that would mimic the sensor inputs and not exceed the time given for us to finish the project.
My inspiration came from this https://github.com/ArztSamuel/Applying_EANNs project, where the developer also used collision as a substitute for real life sensors.
The algorithm works by first initializing a 6 – 36 -36 – 2 Dense neural net with Adam optimizer and a learning rate of 0.003. These numbers were found by simply testing nonstop, 6 in and 2 out are obviously predefined for 6 sensors and 2 actions. I have also noticed the car being stuck in local optima, thus opting to use an extra layer of algorithm called epsilon greedy search. This uses a number between 0 and 1 (epsilon) and gradually decreases it, my implementation starts at epsilon = 1 which means at first the AI is assumed to know nothing and simply take random moves to fill some memory, which is intuitive in my opinion. As epsilon decreases the AI starts to learn and exploit the greedy solutions it finds, the gradual decrease allows the AI to explore the entire track with all possible options. I also had a way for the car to cross these so-called reward gates that would give it a fixed amount of rewards, to guide it a little bit, but it

was completely unnecessary and did not show any significant improvement.
At the beginning I had a slow learning rate and a very slow decaying epsilon which made the AI train slow, however after tweaking them a bit I was able to increase the decay rate of epsilon and increase the learning rate of the neural net which resulted in a faster learning without sacrificing the quality.
For the actual simulation, I coded the entire thing from scratch since openAI did not have any options that suited my interests regarding the sensors. I have used a library called pygames. One of the downsides of the entire project was the fact that pygame had no option of omitting rendering, thus the computer renders every single attempt, slowing down the computation in general, however this is still a much better trade off compared to openAI which had an option to disable rendering, but was still about 70-80% slower. At first finding optimal hidden layer sizes for the neural net were difficult, but after some research I noticed that similar problems would multiply their input by 6. I tried doing 36 layers and it worked better than expected, there was one training session where it learned to do the entire track in 44 simulations (it usually takes about 100, and it used to take over 500 simulations before changing epsilon decay, learning rate and the size of the hidden layers).
When it comes to the actual Q learning part, it is standard using this formula

$$\dot{Q}(s,\dot{a}) = r \quad + \beta(\max(Q(s', a')))$$

I found a popular discount standard of 0.95 to work very well with my model. (The discount is the amount that future reward is weighted with, this means the AI will look into doing an action that gives big reward sooner rather than later).



In the picture above, is the general skeleton of how the algorithm works.

It starts of by looking at the starting state, passes it through the neural net and takes an action based on the output. The simulation simulates that action and returns a reward along with a new state. And the cycle repeats as the neural net adjusts weights based on the received reward.

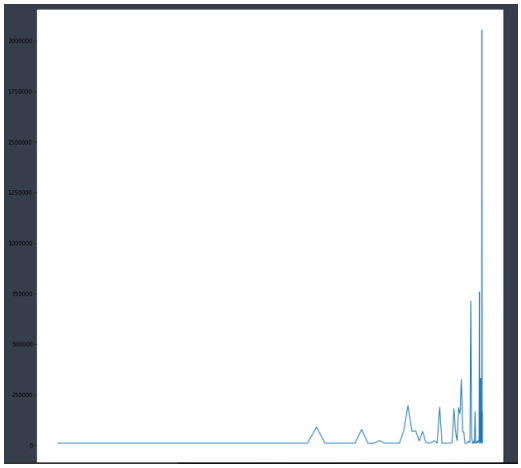
Software

Libraries used	Code written by me
Keras – basically an alternative tensorflow	Game engine – All the equations for turning, moving, how lines are drawn were written by me.
Shapely – simply library used to handle some geometry equations such as line collision	CarAI class – a class that handles the forward propagation of the sensors data. It takes in all the hyperparameters and
Pygame – used to create/render the game and the entire simulation.	Draw – draw function to make quick tracks by drawing on the screen
Deque – a data structure that automatically handles the AI's memory.	Collision – Handling everything in case of collisions.
	Everything that was not part of a library was coded by me.

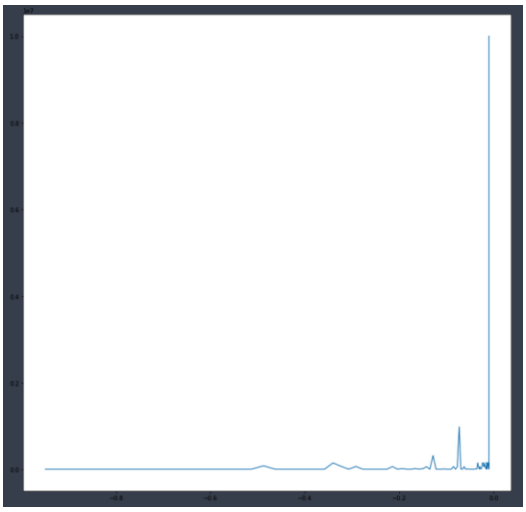
Experiments and Evaluation

At first, I have used gym specifically its car-racing simulator, however due to the convolution done beforehand, everything was slow and not responsive. Unfortunately, I did not plot any graphs, as I was just testing things out.

I then started working with my own implementation and started out with too many variables and a slow learning rate. As discussed previously, the back sensors did not contribute at all and simply confused the AI, the reward gates were supposed to be helpful guiding beacons for the car, however the car did not seem to care about them, I believe it had to do with how little the reward gate's reward was compared to just staying alive. All graphs provided are score vs epsilon, which represents how the score improved as the AI trained



Notice the exponential rise in score as the car figures out that it needs to keep its distance from the walls. Also note that this model did not complete the track and was stuck eventually reaching the end of the entire simulation. I believe it was due to the back sensor messing up everything. After tweaking the learning rate, epsilon decay and taking out unnecessary sensors. Here is the result



You can see that once the car figures out how to complete the lap, it goes until the end. Also notice the wide oscillations in the beginning compare to how narrow they get. This is due to the epsilon decreasing which means less random movements and more stable execution. This model successfully completed the lap and reached the maximum rewards possible.
One of the problems I found in this approach was the fact that epsilon was not so reliable. Sometimes a very bad set of random numbers would cause the AI to not really explore and keep doing a greedy algorithm.

Discussion and Conclusion

Overall the project was a success for me. I was able to first learn to simulate environments on my own. Apply the Q learning algorithm and have a completely different understanding of hyperparameters compared to simply reading theory. There were many things that had to be taken care of and I was constantly finding solutions to arising problems, and quick ones, because I couldn't afford to start over.
I think Q learning itself is great for games with finite outcomes – however Deep Q learning is truly powerful reinforcement learning algorithm that can easily be how robots would learn in the future. The idea of simply letting it learn on its on is incredible and game breaking compare to old school methods of collecting ton of data and feeding it a simplified version until it finds a pattern in that data and is able to identify the pattern given some unknown data.
When I started with Q learning in theory I expected it to be not as easy as it looked on the paper (just a hunch) and I was correct, the few dozens of optimizers and loss functions turned the neural network part into a full on testing problem, given the amount of resources I had at my disposal it was impossible for me to test all of them, so I had to

research some in theory and see which ones would handle the type of data I supplied my car best. This alone introduced me to many optimizer algorithms that I never knew about. Such as RMSprop, Sigmoids and etc.
I initially expected it to take about 12 hours of training before the car could drive the entire track, however to my surprise it only took about 15 minutes of training on average. I also never expected the implementation of the reward system to be the hardest part of Q learning. Trying to understand how I should tell the computer that it is doing a good job is a problem in of itself. I think my idea of forcing the car to drive forward and giving it points encouraged to stay alive, so it worked out.
The biggest limitation to this approach is probably the fact that it works by 1 generation at a time. I realized how efficient evolution-based models are due to the sheer number of generations each time. However, these methods don't use Q tables, but rather mutate weights directly. I think an interesting algorithm that I haven't seen yet would be to combine all 3 together a Deep Evolution Q learning, combining Q learnings intuitive approach to with Evolution's fast and greedy solution findings, it can be really powerful. I realized this too late unfortunately.

Also, the entire window constantly rendering was causing some slowness in the algorithm. I would try to find an alternative library that supports turning rendering off option. In a lab I would most definitely go straight to 3D simulation and try to add more complexity to the model as it learns, such as other cars, obstacles, humans and slowly turn into a real-life driving simulator. Once we have the brains of the car, all is left is to map the outputs to car controls and map various sensors to the input.