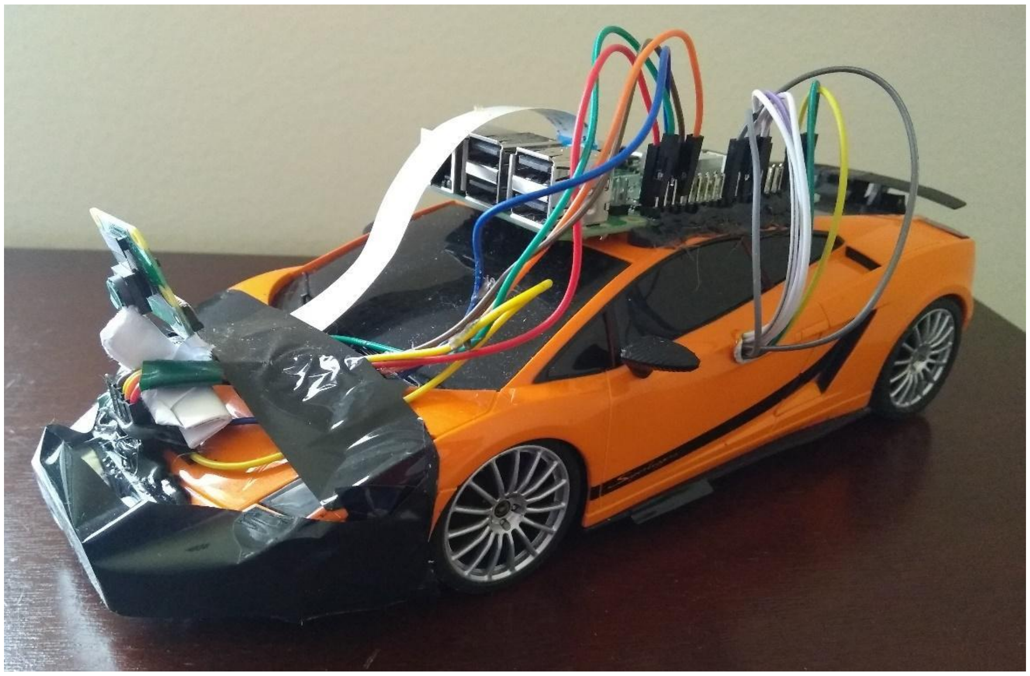
ENPM808F – Robot Learning
Final Project Report

Deep Q-Network (DQN) Based Lane Following Toy Car



Dipam Patel | 115809833
Shivang Patel | 115504950

Instructor: Professor Donald A. Sofge

UNIVERSITY OF MARYLAND

Abstract

This project is an attempt to present a reinforcement learning approach using Deep Q-Networks to steer a toy car on a real track made of white road and black lanes. An action-based reward function is proposed, which is motivated by a potential use in real world reinforcement learning scenarios. We provide a general and easy to obtain reward: the distance travelled by the car without the supervisor taking control. Compared to a naive distance-based reward function, it improves the overall driving behavior of the car agent. From randomly initialized parameters, our model is trying to learn a policy for lane following using a single monocular image as input. We use the deep q-network algorithm, with all exploration and optimization performed on the computer connected to the raspberry pi. This demonstrates a new framework for autonomous driving which moves away from reliance on defined logical rules and mapping. We discuss the challenges and opportunities to scale this approach to a broader range of autonomous driving tasks.

Keywords- Self-driving, lane following, deep q-network, neural network, q-learning

Contents

TITLE OF THE PROJECT……………………………………………………...1

ABSTRACT……………………………………………………………………….2

1. INTRODUCTION 5

1.1 Overview of Autonomous Vehicles……………...5

1.2 Recent Development……..5

2. BACKGROUND……………………………………………………………...….6

2.1 Reinforcement Learning……………………...6

2.2 Q-Learning………..……...7

2.2.1 Exploration and Exploitation……………………………………………………8

2.3 Deep Q-Network……..8

2.3.1 Experience Replay………………………………………………………………9

2.3.2 Periodically Updating the Q-values ………………………………………….9

2.3.3 State Representation in DQN …………………………………………………10

3. DESCRIPTION….……………………………………………………………….11

3.1 Description of Project..…………………………………………………………11

3.2 Description of Toy Car …………………………………………………..…..11

4. ARCHITECTURE OF DEEP Q-NETWORK (DQN) ALGORITHM……………13

4.1 Pre-processing Part...13

4.2 Problem of Temporal Limitation..13

5. IMPLEMENTATION & RESULTS...15

5.1 Experimental Setup 15

5.2 Software Setup………………………………………………………………………15

5.3 Experiments ………..16

5.3.1 Training ……………………………………………………………………16

5.3.2 Testing ……………………………………………………………………...17

5.4 Implementation Issues 18

6. SUMMARY …..……….………...19

7. FUTURE WORK..20

8. REFERENCES 21

1. INTRODUCTION

Machine Learning has changed the course of AI in recent decade. Even though machine learning concepts are half a century old, they are being used widely today due to advancement in computing technology. ML has changed the worth of data and due to which it is now considered as an important raw material in the modern industries. These advancements has motivated the innovators to heavily invest in the autonomous market which now seem promising business. This eventually increased the research and development in autonomous vehicles industries.

1.1 Overview of Autonomous Vehicles

Most approaches in Autonomous Driving rely on external mapping infrastructure instead of understanding the local scene. The generality of reinforcement learning makes it a useful framework to apply to autonomous driving. Reinforcement Learning demonstrates a new framework for autonomous driving which moves away from reliance on defined logical rules, mapping and direct supervision.

1.2 Recent Development

Autonomous driving is a topic that has gathered a great deal of attention from both the research community and companies, due to its potential to radically change mobility and transport. Broadly, most approaches to date focus on formal logic which define driving behaviour in annotated 3D geometric maps. This can be difficult to scale, as it relies heavily on external mapping infrastructure rather than primarily using an understanding of the local scene.

In order to make autonomous driving a truly ubiquitous technology, robotic systems which address the ability to drive and navigate in absence of maps and explicit rules, relying - just like humans - on a comprehensive understanding of the immediate environment while following simple higher-level directions (e.g., turn-by-turn route commands) needs to be advocated. Recent work in this area has demonstrated that this is possible on rural country roads, using GPS for coarse localization and LIDAR to understand the local scene.

In the recent years, reinforcement learning (RL) – a machine learning subfield focused on solving Markov Decision Problems (MDP) where an agent learns to select actions in an environment in an attempt to maximize some reward function. We argue that the generality of reinforcement learning makes it a useful framework to apply to autonomous driving.

2. BACKGROUND

2.1 Reinforcement Learning

A reinforcement learning problem as a sequential decision-making problem under uncertainty. This can be written down in the form of a Markov decision process. It has state space which is denoted as S, a set of actions of the agent A, transitions from state $s \epsilon S$ to state $s' \epsilon S$ with action $a \epsilon A$ at time $t \epsilon T$ denoted as a_t and which is denoted as $P(S_{t+1} = s' \mid s_t = s, a_t = a)$. Next to that there is a reward function $R(s, s', a)$ that returns the reward which is given from s to s' with action a.

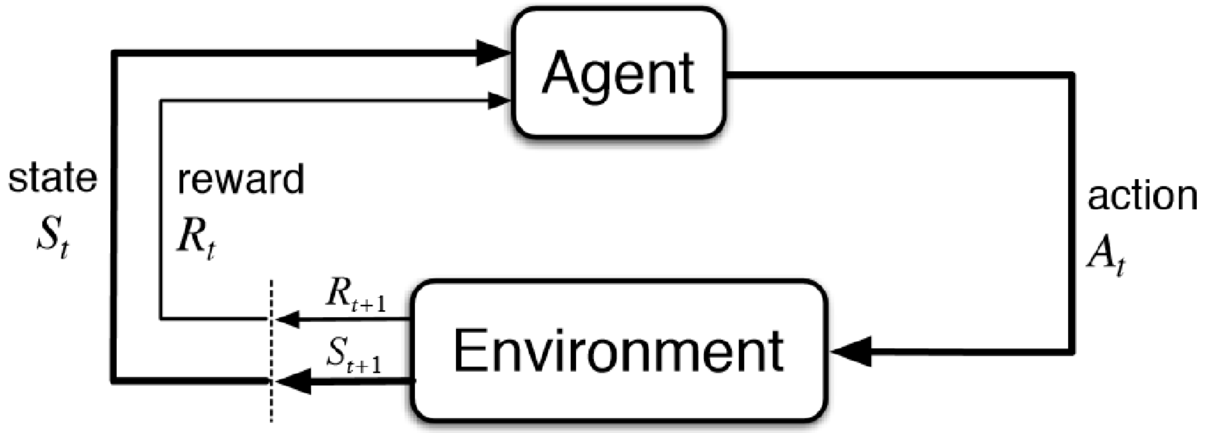


Figure-1 Reinforcement Learning

There are two different types of models which will determine the action an agent takes. A model-based algorithm learns the transition probability from each state-action pair. A model-free algorithm relies on a trial-and-error strategy, where it keeps updating its knowledge. A policy _ denotes a method by which the agent determines the next action based on the current state. Instead of a reward function, we define a value function $V\pi(s)$ that maps the current state s to the expected long-term return under policy π. In the next section, we will go deeper into this value function. But first, we need to explain that there are two different types of policies, on-policy and off-policy. An on-policy agent learns the value based on its current action A derived from the current policy, whereas its off-policy counterpart learns it based on the action $A0$ obtained from another policy.

2.2 Q-Learning

Ideally, the reward function can be found, but in most cases that is infeasible. Due to randomness and other factors which cannot be influenced by the actions, an exact reward function cannot be found in most real-life problem settings. This is why a value function is introduced. It calculates the expected reward based on the state and action. This value function needs to be learned based on the collected data. There are multiple ways of learning a value function and one way is known as Q-learning. For every state-action pair, it has a Q-value which is the reward used to provide the reinforcement and can be said to stand for the quality of an action taken in a given state. Q-learning is an off-policy and model-free reinforcement learning algorithm. The following equation is the rule how the value is updated of a state-action pair.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{A_t} Q(S_{t+1}, A_t) - Q(S_t, A_t)]$$

After every time step *t* the Q-function can be updated by the new information it obtained. The first part of the function contains simply the old value of the Q-value. The learning rate α is a value between 0 and 1 and determines to what degree new information influences the Q-value. This α is multiplied by the actual reward, R, added to the estimate of the optimal future value $Q(S_t, A_t)$. This estimation is adjusted by the discount factor γ. The discount factor γ determines the importance of future rewards. It is also a number between 0 and 1. When the discount factor is close to zero, it only cares about the current rewards. When it is close to 1 it will try to optimize on the long-term reward. Next, to choosing the appropriate α and γ there needs to be selected initial Q-values for every state-action pair. There are multiple ways to initialise Q-values. How to choose the best approach depends on the problem statement. If exploration is encouraged in the beginning, high initial Q-values are chosen assuming that always the highest Q-value is selected. This is because an update of the Q-value will lead to a lower Q-value. The next time another action will be chosen until the Q-values will converge to their (local) optimum value. When choosing low initial Q-values, it can happen that the exploration will be minimal.

Now we have Q-values that represent our current estimation of the reward given a state and a policy. Next to that, we have a Q-function that is updating the Q-values based on new data. These values are all stored in for example a matrix which has a Q-value for every state-action pair. This is called the tabular implementation of Q-learning. After every step, the Q-function will be updated with all the corresponding effects for the Q-values. The more data it processes the more accurate predictions of the policy will be. The higher the Q-value, the higher estimation of a reward. When all Q-values are low in a certain moment, the Q-function cannot estimate which action will lead to a high reward.

2.2.1 Exploration and Exploitation

Depending on the environments, just performing the action which has the highest Q-values will likely converge to a local optimum. For example, when in the first step a certain state-action pair has a high Q-value, it will never perform another action in that state. To avoid this there are multiple ways to perform exploration. The most implemented way to do this is having a random element when deciding which action to take. Instead of looking at the Q-values a fraction ϵ of the times a random action is performed. With ϵ between 0 and 1 This is called the ϵ-greedy exploration method. Next to the ϵ-greedy exploration method, there are various ways to perform exploration.

A common dilemma in reinforcement learning revolves around balancing exploiting known information and exploring the problem space to find new information. In this research, this will be the main focus of the different algorithms that are being compared. All different algorithms have a different method to deal with this dilemma.

2.3 Deep Q-Network

Before starting to compare the different exploration methods we should first set up an algorithm that will solve the problem with at least one of the different exploration methods. The algorithm we have chosen in this thesis is a deep Q-Network (DQN) as this algorithm has proven to work in different environments. The different exploration methods are all modification of this original algorithm.

Deep Q-learning is a form of the classic Q-learning model. At every step, the state is evaluated and will give a Q-value which approximate the reward of each possible action. Traditionally Q-learning was designed to have a value for every state action pair which called tabular Q-learning. But this is not extendable when the state space is increasing as the possible pairs are growing exponentially. For example, when having 100 different binary variables which define the state space and 10 different actions to take it means that there should be more than $1.26 \cdot 10^{31}$ different Q-values, which also needs to be updated at every time step. Also, a lot of problems do not have binary variables as input space but have more values for a variable or even continues. This kind of problems makes the Q-learning explode and not feasible for a lot of problems. To avoid these limitations a machine learning technique is used to approximate the rewards at every action. Machine learning is applied to makes a supervised model with input state and output action. In this case, a deep neural net is used as a machine learning model, hence the name Deep Q-learning.

The model learns from previous experience in mini-batches, which avoids the model to train after every step. This approach makes sure the algorithm is time efficient and stays feasible, now the only things that will be stored are all the history state-action pairs and the model which are for a deep neural network only the different weights and not all possible state-action pairs of the model. The input of this model is the state-action pairs and is optimized on the obtained reward. The output of the model are the Q-values for the different possible actions.

Next to having the ability to tackle problems with a bigger state space, not only discreet but also continues variables, it is also more scale-able. The method of using a nonlinear function approximate such as a neural network which represents an action-value is known to be unstable or even diverge. There are two main ways to avoid this behaviour. The first one is the use of experience replay 15 and the second one is periodically updating the Q-values. These improvements and others will be discussed in the next sections.

2.3.1 Experience Replay

Every time step the agent stores the obtained data. The data which is stored is $e_t = (s_t, a_t, r_t, s_{t+1})$, with experience, state, action, reward and next state at time t correspondingly. The dataset is defined as $D = \{e_1, ..., e_t\}$. To introduce a new mechanism to remove correlations in the observation sequences and to smooth over the data distribution a sample is taken from D, $(s, a, r, s') \sim U(D)$. This uniform sample is taken when the algorithm is updating the Q-values in mini-batches. The last N experiences are stored which is called the replay memory. The replay memory does not make any disquisition in the importance of the different transitions and overwrite the oldest transitions with the newest when the memory buffer reached N. This is also the case with the mini batches. So, although there can be made some improvements, this solution tackles the main problems.

Prioritized Experience Replay

One improved that can be made to learn more from some transitions than from others is to use prioritized experience replay. This can be done to look at transitions which do not fit well. Instead of uniformly sampling from the replay buffer, there is been looked at the error which is made by the value function. The bigger the error the higher probability to get in the mini-batch. The selection is efficiently done by having a binary tree which has the error for each index of the memory buffer which does not slow down the algorithm by much.

2.3.2 Periodically Updating the Q-values

Another way to improve the stability and especially reduce the observation sequence is to introduce two different networks: Q and $\hat{Q}$. This is called Double Q-learning. At every C number of updates, the network Q is cloned to $\hat{Q}$. This makes sure that the data is not including too recent observations in the Q-values. A direct consequence of that is that the current state is not influencing the predictions of the last couple of states. This works well as $Q(s_t, a_t)$ is highly correlated with $Q(s_{t+1}, a_t)$, so adding a time delay the Q-values will not be influenced by the recent observations.

2.3.3 State Representation in DQN

The state can be represented in various ways when using DQN. It really depends on the available data in the environment. It can be observed data by sensors. This can be all data which represents the current state. Examples are temperatures of sensors, coordinates of certain objects or even screenshots of videos. There are even video games that can learn on the RAM of the game, so the input state is an array with bytes. All data that represents the state space can be helpful to obtain the most optimal Q-values. In this research, we will also look at the pixels of a screen and transform that to a format that can be processed by our DQN, such as an array of RGB-values.

An episode is a complete play from one of the initial state to a final state. In every episode, the same steps will be checked. The most important steps will be explained. First the action will be decided, this is done based on the highest corresponding Q-value or there will be chosen a random action. This action will be performed, and this will be stored in the replay memory. Then the network is trained by performing a gradient descent based on the returned reward.

3. DESCRIPTION

3.1 Description of Project

This project aims to implement the Deep Q-Network (DQN) Algorithm to train a Toy Car to drive by itself while following the lanes. Using only the monocular camera image as input, the car is trained to follow the lanes without any previously collected data for training. The algorithm is rewarded based on the distance travelled by the car before it leaves the lane and is stopped by the supervisor by giving a negative reward. A model-free approach based on the trial and error strategy where it keeps updating its knowledge.

From the randomly initialized parameters, the model is trying to learn a policy for lane following and optimizing it on the go. Raspberry Pi 3B+ was used as an onboard processor on the car to stream frames and perform the actions (forward, left and right) received from the computer which acted as the central processor. ROS was used for communication between Raspberry Pi and Computer. Also, Tensorflow 1.8.0 framework was used to train the convolutional neural network

3.2Description of Toy Car

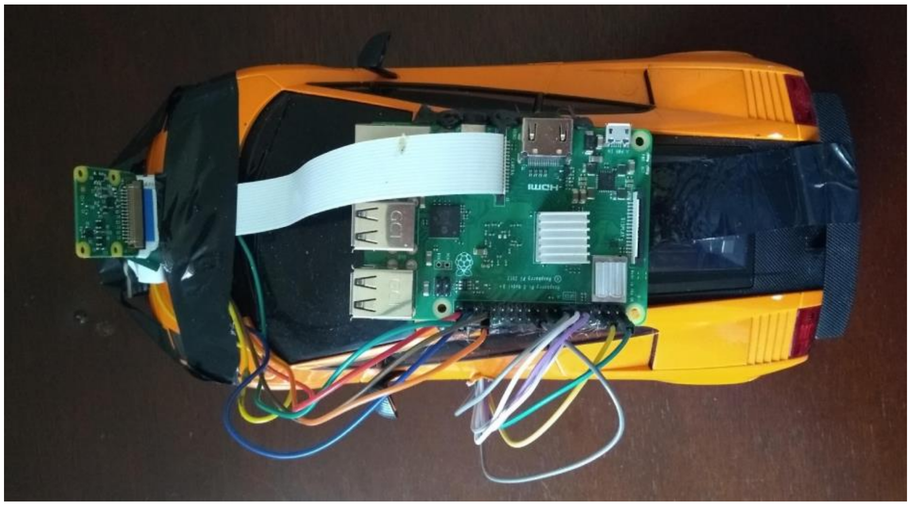


Figure-2 RC Toy Car Model

The hardware used for the project is

- RC Toy Car
- Raspberry Pi 3B+

- Raspberry Pi Camera v2.1
- L293D Motor Driver
- Cables
- Batteries
- Track Details: White Paper as Track and Black Tape as Lanes

4. ARCHITECTURE OF DEEP Q-NETWORK (DQN) ALGORITHM

Instead of just using a Q-table, we implement a Neural Network that takes a state and approximates Q-values for each action based on that state. The Neural Network takes a stack of four frames as input, passed through its network and output a vector of Q-values for each of the three actions. Initially, the agent performs really worse, but with time, it begins to associate frames (states) with best action.

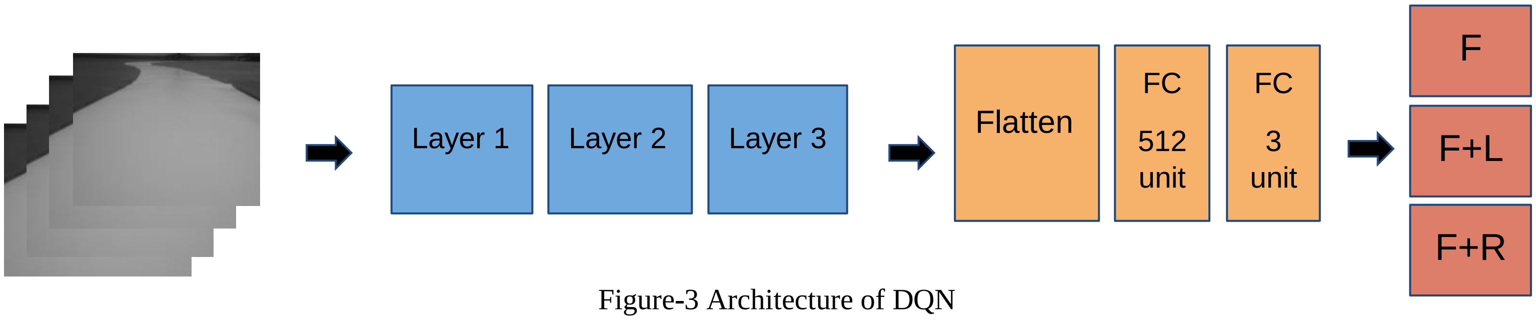


Figure-3 Architecture of DQN

4.1 Pre-processing Part

- ❖ Reducing the complexity of the states to reduce the computation time needed for training
- ❖ Convert RGB to Grayscale as color is trivial
- ❖ Crop the frame from the top to reduce its size and stack four sub-frames together

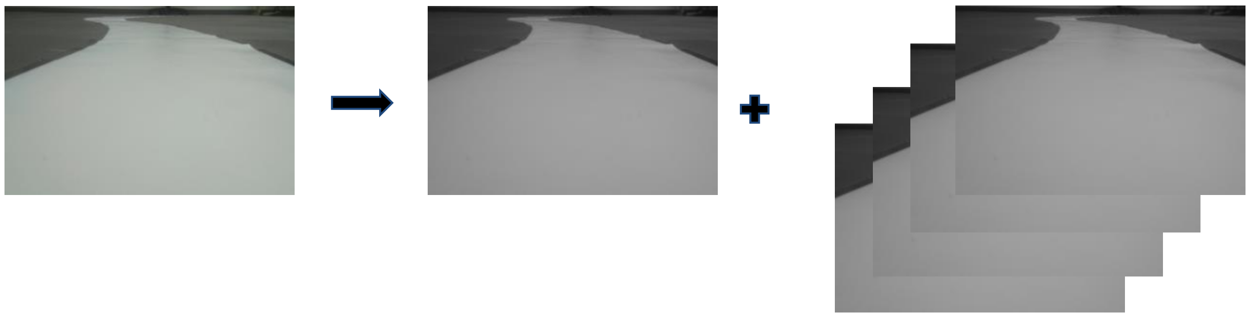


Figure-4 Pre-processing and Stacking in Sub-frames

4.2 Problem of Temporal Limitation

One frame is not enough to have the sense of motion. But, if we add three more frames, we can see where the car is headed towards. Thus, providing a stack of four consecutive images, helps the algorithm determine the direction of motion.

Figure-5 Problem of Temporal Limitation

5. IMPLEMENTATION & RESULTS

5.1 Experimental Setup

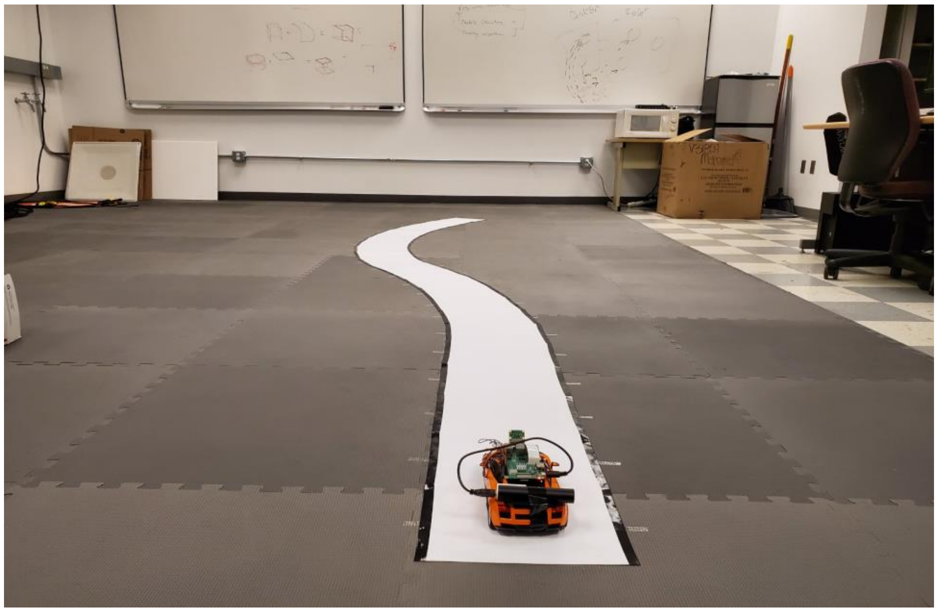


Figure-6 Track with the Toy Car

The white track is laid down in a random shape and the car is kept on the track to allow it to run randomly and explore in the starting.

5.2 Software Setup

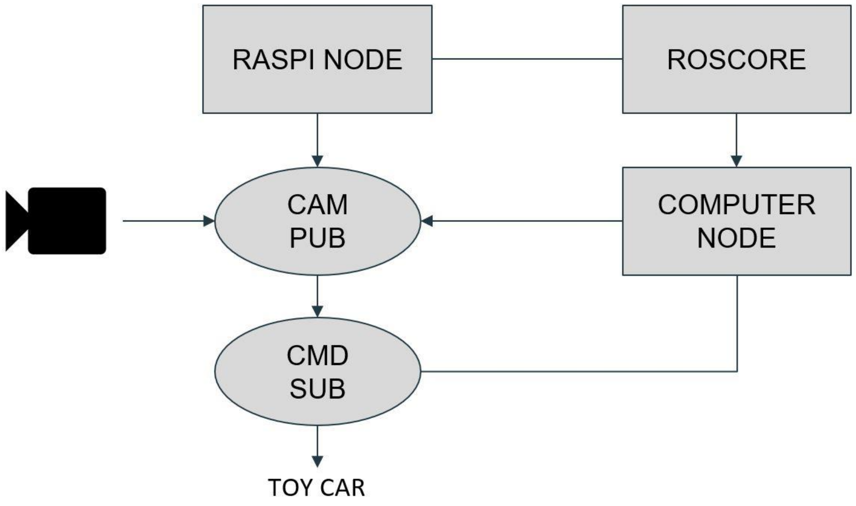


Figure-7 Software Setup & Communication

The raspberry pi has a raspinode which communicates with the computer having an active roscore running on it as well as its own rosnode. The raspinode communicates with the raspberry pi camera and the camera publishes images. The computer rosnode subscribes to the camera publisher and sends the commands to the toy car after processing.

5.3 Experiments

Reward Model

Leaving the Lane:	-10
Following the Lane:	+5
Completing the Lane/Episode:	+10

Insight into Training Params

● State Size	[200, 200, 4]
● Action Size	3 i.e. Left, Right and Forward
● Learning Rate	0.01
● Exploration	1 to 0.01
● Decay rate	0.001
● Gamma	0.95
● Memory size	1500
● Batch size	64

5.3.1 Training

The graph shows a gradual rise in the total reward per episode as the training move towards 800 episodes. Also, some random spikes were observed which can be considered as noise for the graph.

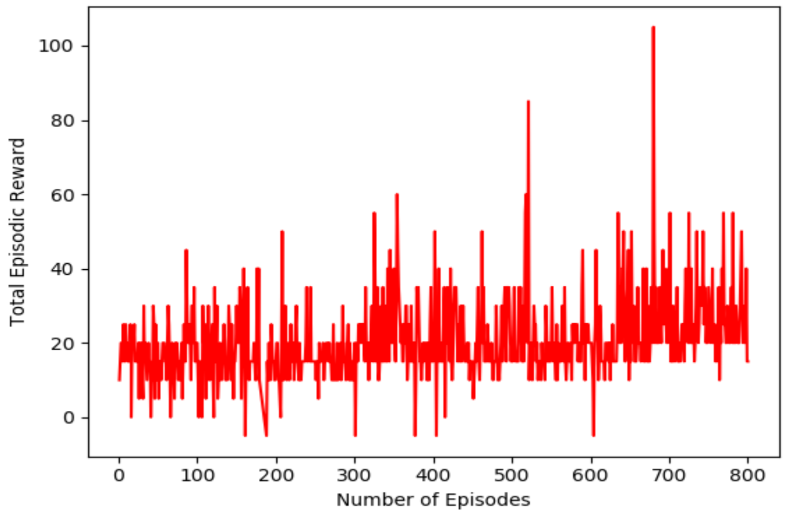


Figure-8 Total Episode Reward v/s Number of Episodes (Training)

Link to Training Video:
https://drive.google.com/file/d/180Qn422pd3R9GoEO2Z3jaWm0X48zOT9E/view

5.3.2 Testing

The graph shows a gradual rise in the total reward per episode as the testing move towards 32 episodes. Also, some random spikes were observed which can be considered as noise for the graph. The testing was performed after every 25 episodes.

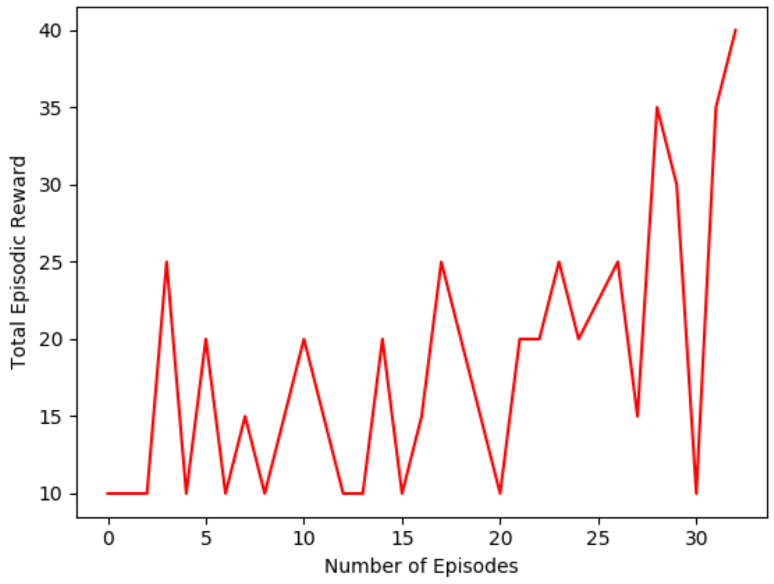


Figure-9 Total Episode Reward v/s Number of Episodes (Testing)

Link to Testing Video:
https://drive.google.com/file/d/1rllwzH8UelCLR_YSGW9ly_s5jMo-VzgR/view

5.4 Implementation Issues

- The variable command delay between the Toy car and Computer
- Computer's physical memory limitation for the Experience Delay

Link to Codes:
https://drive.google.com/open?id=1Iq9u1Ckv_Lp9fQoctYUTdmuSmczXdpai

6. SUMMARY

- Trained an actual Toy Car to follow lane using a DQN Reinforcement Learning Technique.
- Used Experience Replay and Frame Stacking approach for better performance.
- The results show that the unique reward-penalty system greatly improved the learned driving behavior compared to a naive system.
- Due to time limitations and hardware restrictions, we could not have enough timesteps to converge to an optimal solution for any of the different algorithms.

7. FUTURE WORK

- Train a car in a simulated environment based on the camera input and steering angles for a large number of episodes.

- Transferring the trained model to the toy car and testing on the track would give the desired output as the learning policy would be tuned to be optimal for the task.

- Training data would be cheap, auxiliary ground-truth information can be provided with ease, the vehicle can be put into situations that are difficult and more number of episodes can be executed.

8. REFERENCES

1) Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, Amar Shah, “Learning to Drive in a Day”
2) Alex Bewley, Jessica Rigley, Yuxuan Liu, Jeffrey Hawke, Richard Shen, Vinh-Dieu Lam, Alex Kendall, “Learning to Drive from Simulation without Real World Labels”
3) Peter Wolf1, Christian Hubschneider1, Michael Weber1, Andr´e Bauer2, Jonathan H¨artl2, Fabian D¨urr2, J. Marius Z¨ollner1,2, “Learning How to Drive in a Real World Simulation with Deep Q-Networks”, 2017 IEEE Intelligent Vehicles Symposium (IV), June 11-14, 2017, Redondo Beach, CA, USA
4) Christopher Watkin, “Learning from Delayed Rewards”
5) Tim Elfrink, “Performance of Exploration Methods using DQN”
6) Martin Riedmiller, Mike Montemerlo, Hendrik Dahlkamp, “Learning to Drive a Real Car in 20 Minutes”, Frontiers in the Convergence of Bioscience and Information Technologies 2007
7) https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8
8) https://zhengludwig.wordpress.com/projects/self-driving-rc-car/
9) https://towardsdatascience.com/build-your-own-self-driving-toy-car-ad00a6804b53
10) https://github.com/bdjukic/selfdriving-robot-car
11) https://blog.floydhub.com/toy-self-driving-car-part-one/
12) https://github.com/RyanZotti/Self-Driving-Car
13) https://medium.com/@jonathan_hui/lane-keeping-in-autonomous-driving-with-model-predictive-control-50f06e989bc9