Deep Q-Learning for Self Driving Cars to Avoid Obstacles

Aditya N. Govardhan, Lekhavarshini Selvam, Ronik Dhakar

Abstract: Reinforcement learning is a learning paradigm where an agent interacts with the environment and learns more about the environment by interacting with it. The goal of this interaction is to maximize the reward function. However, formalizing such problems can turn out to be complex and thus a basic framework of modelling a reinforcement learning problem is discussed in this paper. When the agent is unaware of the environment, the designed agent is known as model free agent since it doesn't have information about the environment. One of the famous techniques for solving model free control problems is Q-learning which is studied in depth. Accordingly, for cases of practical implementation with large number of states, deep Q-learning is used, which is also discussed in this paper. These concepts are implemented practically using a self driving car problem which has a goal of driving without crashing into any obstacles.

***Index Terms*—Reinforcement Learning, Agent, Environment, Reward, State, Action, Value Function, Deep Q-learning**

I. Introduction

Reinforcement learning is one of the important areas in artificial intelligence problem solving techniques. In this paper, basic concepts and terminologies will be discussed and how these concepts are connected, will be explained. One of the various techniques in solving reinforcement learning problems is Q-learning. This paper discusses in depth how Q-learning works and how it is practically implemented using deep Q-learning. To establish these concepts and the working of deep Q-learning, a demonstration of a self driving car, with an objective of travelling without crashing into obstacles, is discussed. Correspondingly obtained results of performance are presented.

Reinforcement Learning

Reinforcement learning has an agent interacting with an environment. The agent receives a state and action is taken in that state in a way which optimizes the reward from the environment.

The important aspect in RL is the reward function denoted as R_t which is a scalar feedback signal from the environment. At every time step t, the random variable R_t is evaluated which tells us as to how well the agent is performing at that time step or what is called the state. And the goal of the agent is to maximize the cumulative reward obtained.

RL required sequential decision making where the goal is to choose actions that maximise the total future rewards. It is also necessary to note that the actions taken have a long term consequence and hence sometimes there is a delay in receiving the rewards. We also should sacrifice certain rewards in order to get a larger reward at a later stage.

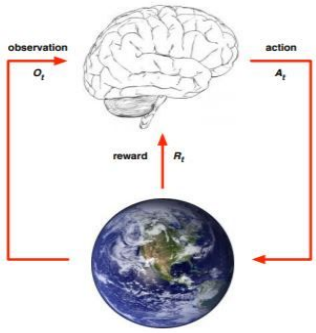

Fig. 1. Components of RL

The brain is the agent here. For any action taken on the environment, we obtain observations and the rewards. We have no control over the environment and what rewards we may get. The only we interact with the environment is through performing some action.

History is the sequence of observations, actions and rewards and the state is the information used to evaluate what happens next. Hence we can say that the state is a function of history.

$$S_t = f(H_t)$$

Environment State

Denoted as S_t^e is the information contained in the environment which determines the course of actions. It's basically to say what state the environment is in.

Aditya N. Govardhan, Lekhavarshini Selvam and Ronik Dhakar is with the Arizona State University, Tempe, AZ 85281 USA (e-mail: Aditya.Govardhan@asu.edu, lnolast3@asu.edu, rdhakar@asu.edu).

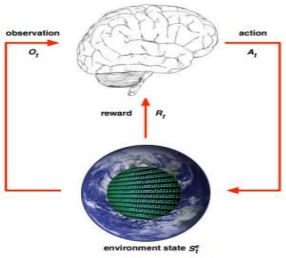

Fig. 2. Environment State

It is just a set of numbers or the information which determines what happens next from the perspective of the environment. The agent does not have any information about the environment but only of the observations from the environment. Even if we can see the information contained, it may not be relevant.

Agent State

Denoted as S_t^a is the set of numbers or the information contained in the algorithm and these numbers are used to determine the next action.

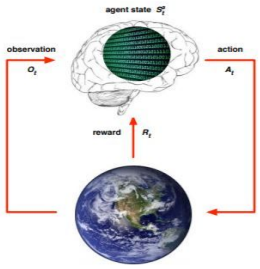

Fig. 3. Agent State

It is our decision to store what observation we get from the environment and what to use to build out RL algorithm.

Information State

The state can be mathematically defined and this is known as the information state or the Markov State which contains every information obtained from the history. The Markov Property can be written as

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1,...,S_t]$$

which shows that the next state predicted from all the previous states can simply be evaluated using just the current state and it gives the same characterization of the future. Basically, given the present we can ignore all the past information to evaluate the future. Hence, the current state is a sufficient statistic of the future.

We need some description about the environment that the agent interacts with and that is given by the Markov Decision Processes (MDPs). The environment can be completely observed and analysed for the algorithm where the current state describes the process. MDPs can be used to solve almost all the RL problems. Optimal control that we aim we achieve is mainly used for continuous action states and even partially observable environment can be converted into an MDP problem. Bandits are one state process where an agent takes an action in an environment which gives a reward. This is also an example of MDP.

To see how Markov Process works, we can imagine that there is a state *s* and the next state *s'* and the probability of the state change from s to s' can be defined as

$$P_{ss}^{'} = P[S_{t+1} = s' \mid S_t = s]$$

This can be summarised in a matrix form for all the states described by the environment which is known as the state transition matrix. The dynamics of a system can be completed shown by the state space and transition probability (S, P) without the action and rewards being defined.

Another is the Markov Reward Process (MRP) which is basically the Markov process with some values judgement. It takes into account the rewards being accumulated along the process. This is a space of (S, P, R, γ) where R and γ denote the reward function and discount factor respectively.

$$R_s = E[R_{t+1} \mid S_t = s] \text{ and } \gamma \in [0, 1]$$

The return G_t is the goal that we want to achieve. It is the total discounted reward from any time step *t*.

$$G_t = R_{t+1} + \gamma R_{t+1} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The model that we use isn't a perfect one. We just use the Markov Process to build the environment but we do not know if it would work. Hence, we use the discount factor to trust the model to give a huge reward by trial and error method of choosing actions that could work.

Monte Carlo and Temporal Difference Learning

For realistic constraints, we give up the idea that we have the information on how the environment works and what feedback it would give and define a model free prediction which works for unknown MDPs.

Monte carlo is an ineffective but straight-foward method of solving a model free prediction problem. It learns from the entire set of episodes. Whereas Temporal Difference reinforcement learning method learns from the incomplete set of episodes that is by bootstrapping. We do not estimate from the whole episode but by taking a partial step and making a guess what it would take to reach the goal. So at each step, it keeps updating its projected value function as to how long it would take to the reward.

Policy and Value Function

The agent's behaviour is termed as policy. That is for some state *s,* it can be mapped to an action *a*. The policy indicates if the specific action gives maximum reward or not in that state.

Value function predicts the future rewards. It is a function of state or state-action pair that estimate agent in a state or

perform action in a given state. We can define the two value functions - state value function and action value function.
State value function indicate d as V_π tells us how good any given state is for an agent following policy π. It is the value of the state under π.

$$V_\pi(s) = E[G_t \mid S_t = s]$$
$$= E[\sum_{k=0}^{\infty} \gamma k R^{t+k+1} \mid S_t = s \ ]$$

Action value function q_π tells us how good it is for an agent to take any action in any given state following policy π. It is the value of an action under π.

$$q_\pi(s) = E[G_t \mid S_t = s, A_t = a]$$
$$= E[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \ ]$$

Optimal Policies

Basically, the RL seeks to obtain more return to the agent than all the other policies. We can say

$V_\pi(s) \geq V_*(s)$ if and only if (s) ≥ (S) for all s ϵ S.

Therefore, we define the optimal state value function which is the maximum value of state under all policies. It is the largest expected return achievable by any policy π for each state.

$$V_*(s) = max_\pi \ V_\pi(s)$$

Similarly, the optimal action value function can be defined as the maximum value of the state-action pair under all policies.

$$q_*(s,a) = max_\pi \ q_\pi(s,a)$$

The function $q_\pi(s,a)$ is the Q-function.

Q-learning

It is RL technique for learning the optimal policy in Markov Decision Process. It enables choosing an action using behavior policy as well as actions that could be taken to achieve the target policy. The objective is to find a policy that is optimal. That is the expected return of all successive time steps is the maximum achievable. We learn by finding the optimal policy which is in turn by learning the optimal q values for each state-action pair.

$$V_*(s) = E[r(s, \pi^*(s))] + \gamma E_{s'|\pi^*(s)}[V_*(s')]$$

Here the value of the state is the sum of the expected immediate reward what we get for the state if we take the perfect policy and the discounted value of the next state.
Defining a similar function over a state- action pair instead of just the state, we have the Q-function which is the sum of the expected value of applying action *a* to state *s* and expected value of following optimal policy of the state we end up in to state s.

$$Q(s,a) = E[r(s, a)] + \gamma E_{s'|a}[V_*(s')]$$

Q-Learning works in the following manner. The algorithm iteratively updates the q value for each state action pair using the Bellman equation until the q-function converges to the optimal q-function. This is called value iteration.

$$q_*(s,a) = E[R_{t+1} + \gamma max_{a'} q_*(s', a') \ ]$$

We can iteratively compare the loss between the q-value and the optimal q-value for the state-action pair. And the goal is to reduce the loss.

$$q_*(s,a) - q(s,a) = \text{loss}$$
$$E[R_{t+1} + \gamma max_{a'} q_*(s', a') \ ] - E[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \ ] = loss$$

To see how the q-value gets updated, a learning rate is defined as $0 \leq \alpha \leq 1$. The learning rate tells us how quickly an agent abandons the previous q-value for new q-value for a given state-action pair.
So the new q-value is calculated using this formula -

$$q^{new}(s,a) = (1 - \alpha)\, q(s,a) + \alpha(R_{t+1} + \gamma\, Q(S_{t+1}, a'))$$

which is the weighted sum of the old q-value and the learned q-value.

On-Policy and Off-Policy learning

On-policy learns about the policy π from the experienced samples of π that is it learns from the policy that we are following.

Q-learning is based on Off-Policy learning. While following behaviour policy $\mu(a|s)$, we focus on evaluating a target policy $\pi(a|s)$ so as to compute $V_\pi(s)$ or $q_\pi(s,a)$. It learns from different components in the environment -

- By observing human behaviour
- It re-uses experience from old policies $\pi_1, \pi_2 ... \pi_{t-1}$.
- It learns an optimal policy while exploring the states effectively.
- It may also learn multiple policies while following a single policy. We can figure out different behaviours just from one stream of information.

Off-Policy control with Q-learning

The target policy π is a greedy policy with respect to Q(s,a) that is it allows both the behaviour and the target policies to improve. It evaluates what it considers to be the best policy.

$$\pi(S_{t+1}) = argmax_{a'} Q(S_{t+1}, a')$$

Initially when an agent starts, the q-value is taken as zero and it does not know which action is better. Hence we need to understand the trade-off between exploration and exploitation. This helps us understand how the agent takes its first action and how it chooses its action in the further states. Exploration is referred as the act of finding out information by exploring the environment. And exploitation is the act of exploiting the information about the environment which will be used to maximize the return. To find a balance between the two, an epsilon greedy strategy is used. We assume that the agent explores the environment and not exploit at the beginning i.e $\varepsilon = 1$. As it learns about the environment, the ε reduced as the agent becomes greedy and tends towards exploiting the environment. It is a

way of selecting random actions with uniform distribution from a set of available actions. Using this policy we can select epsilon-1 probability that gives us maximum reward in any state.

Substituting the target policy in the updated new q-value gives us -

$$q^{new}(s,a) = (1-\alpha)\, q(s,a) + \alpha(R_{t+1} + \gamma\, max_{\alpha'}\, q(s',a'))$$

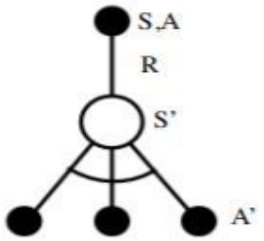

Fig. 4. Explaining Q-learning using SARSAMAX

This is a simple representation of Q-learning called SARSAMAX. In a particular state, an action is performed which gives a reward and creates a new state and new possible actions. We further choose the action that gives the maximum reward.

Deep Q-learning (DQN)

If there is an environment with 10000 states and 10000 actions per state, the Q-table will be a million cell table. The memory required to store the table and update it would be very large and the time taken to evaluate the state-action pairs would be unrealistic. Hence, we make use of deep Q-learning to evaluate the q-table to work with. It is a neural network algorithm with multiple hidden layers that gives a vector of action values Q(s, .; θ) as an output for a given state s where θ is the input parameters to the neural network. The comparison between q-learning and deep q-learning can be seen from the following illustration -

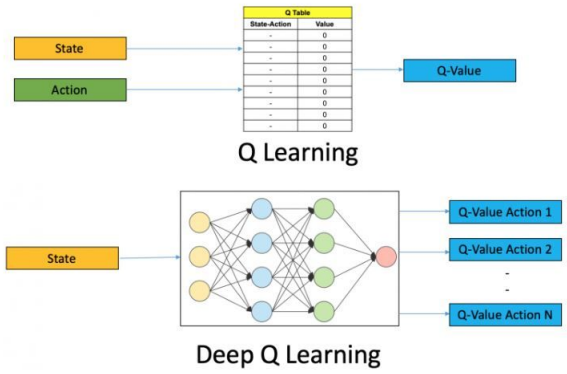

Fig. 5. Comparison of Q-learning with Deep Q-learning

Mnih et al. (2015) proposed that the important elements in a DQN algorithm is the use of target network and experience replay. The target network has parameters θ^- which is similar to the online network but at every τ step, the parameters are copied which gives us $\theta_t^- = \theta_t$. DQN uses the target is now -

$$Y_t^{DQN} \equiv R_{t+1} + \gamma\, max_a\, Q(S_{t+1}, a; \theta_t^-)$$

This contains a memory bank that updates the network. The transitions observed in an experience replay are stored for a short time which then gets sampled uniformly. Hence the performance of the algorithm is improved by both, the target network as well as the experience replay. The components of DQN combined with RL looks like this

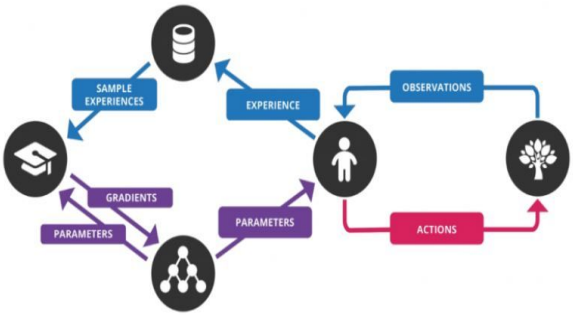

Fig. 6. DQN integrated with RL

II. Avoiding Obstacles by Self Driving Car

Agent and Environment Setup

To gain a practical understanding of reinforcement learning problem in general and deep Q-learning in particular, following reinforcement learning agent is implemented. A self driving car is moving in a rectangular area bounded with walls by all four sides. This area contains two types of obstacles. Three static big obstacles representing house appliance or pillars, and one dynamic small obstacle representing a kid or a pet. For the sake of overfitting mitigation, the big obstacles are not entirely static, but move with a small speed. A self driving car is moving in this rectangle with an objective of not colliding with the moving obstacles or the wall. Fig. 7. is a time snapshot of the setup.

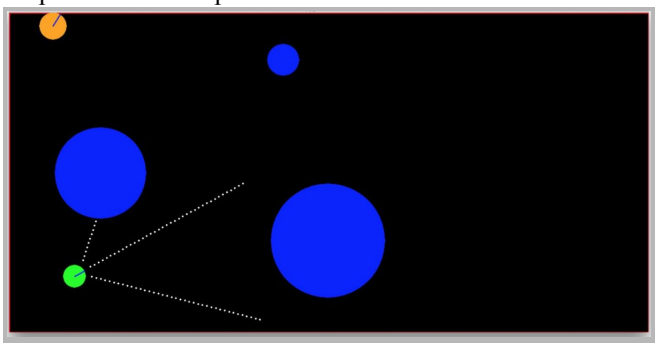

Fig. 7. Frame snapshot of the setup

The green colored circle represents the self driving car

while the blue colored circles represent the semi-static obstacles. The orange colored circle represents the dynamic obstacles. The thin red colored boundaries represent the walls.

The three dotted lines represent three sensors attached to the self driving car. It takes the decision of turning based upon the inputs provided by these sensors. The sensors constantly give feedback of distances measured by them and thus determine the action value. This mechanism is explained in detail in reinforcement modelling.

Setup Development

The language used for coding is Python and its associated libraries. Primarily, Numpy is used for carrying out matrix calculations while Matplotlib is used for graph plotting. The physics engine used for development of environment and agent is Pymunk while the graphics are developed using Pygame. For development of deep learning neural network, Keras is used.

Programming Language	Python
Deep Learning Library	Keras
Physics Engine Library	Pymunk
Graphics Library	Pygame
Plotting Library	Matplotlib

Table 1. Language and packages used

Reinforcement Learning Problem Modelling

The described setup is modelled as a reinforcement problem as follows. Here the agent is the self driving car itself, while the environment is represented by the wall, three semi-static obstacles and one dynamic obstacle. The goal of the agent is to roam in the environment as long as possible without crashing. Intuitively, the reward should increase if the self driving car doesn't crash. Thus to cater to this intuition, the reward is kept as the sum of distances measured by three sensor values - 3, when the car doesn't crash and -500 when the car crashes.

The state is determined is a three element vector determined by the readings measured by three sensors. Thus the state varies with varying distance from obstacles. The self driving car can take three actions: moving left, represented by 0; moving right, represented by 1; and continuing on the current path, represented by 2. These actions are influenced by the current state of the agent as expected. This modelling is summarized in Table 2.

As explained in the introduction, given a state and action, deep Q-learning action helps us determine the next state and action. The following section explains how deep Q-learning is implemented for this reinforcement learning problem.

Goal	Drive car avoiding obstacles
Agent	Self driving car
Environment	Four static walls, three semi-static obstacles and one dynamic obstacle
Reward	-500, if crashed Sum of sensor readings - 3, if not crashed
State	Normalized readings from three sensors
Action	0 - turn left 1 - turn right 2 - continue path

Table 2. Meaning of the RL terms for the given problem

Deep Q-learning Implementation

Deep Q-learning is implemented in this setup using a neural network with four layers. The input layer to the neural network is the state array of size three and the output layer is the Q values for each of the three actions. The first hidden layer is of size 164 nodes while the second hidden layer is of size 150 nodes. For training, the state is fed to the neural network in batches and corresponding batch of Q values is obtained. For prediction, the current state is fed to the neural network, which gives an output of Q values. The action with the maximum value of Q is taken, essentially capturing the essence of Q-learning.

One important aspect to be considered in reinforcement learning problem while training is exploration versus exploitation. Exploitation determines how strongly the training be focused towards optimization of given strategy. Exploration determines how much amount of new strategies should be explored. While exploitation can lead to local minima, exploration can help find global minima. In this implementation, the exploration versus exploitation is determined by a parameter known as epsilon (ϵ). It starts with one and decreases to zero over the period of training, updated after every crash. High value of ϵ indicates exploration while low value indicates exploitation. Thus by the end of the training period, the implementation is more focused towards exploitation.

Results

The self driving car is trained for one lakh iterations or one lakh frames. In each frame, the obstacles move a certain distance and so does the self driving car. At each frame, the action is determined using deep Q-learning neural network. The reward is calculated for each iteration. In this implementation, the term episode is defined as the number of frames from start of motion till crash.

Thus the cumulative reward in each episode is a good indication of how well the neural network is learning. fig. 8. represents the cumulative reward at each frame. While the self driving car is not colliding, the cumulative reward is increasing, as soon as it gets an idea that it is coming close to an obstacle, the cumulative reward bends down for a small time and drops down to zero when it crashes.

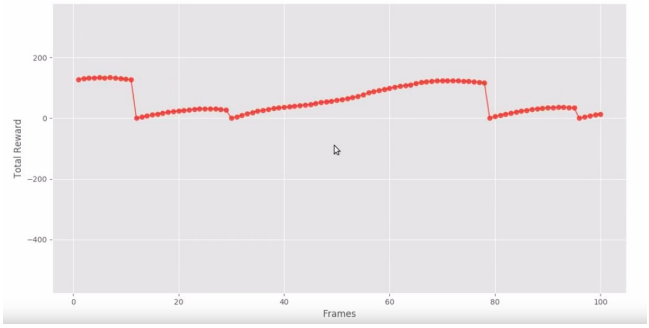

Fig. 8. Cumulative result vs frames

Similarly, the distance covered in each episode is supposed to increase in general if the agent is learning the environment well. As it can be seen in fig. 9, the averaged distance over ten episodes against episodes, increases in general. At the end when the exploitation increases, the averaged distance increases at a greater rate.

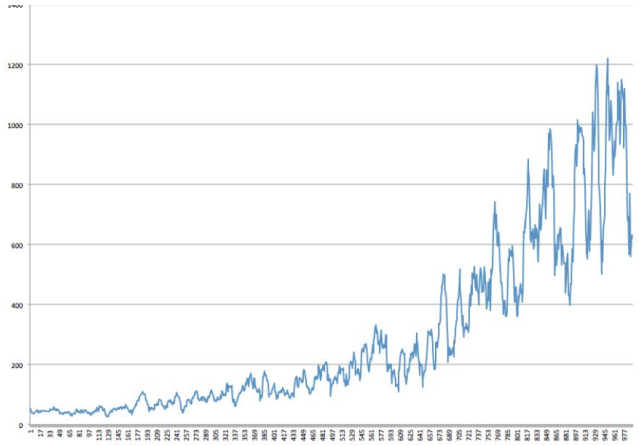

Fig. 9. Average distance vs episodes

Finally, since the neural network is learning constantly and converging towards the optimal behavior, the loss function is expected to decrease as can be seen from fig. 10.

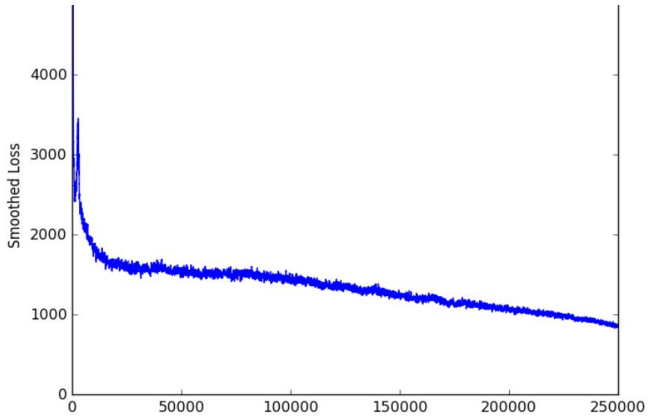

Fig. 10. Smoothened loss vs frames

III. Conclusion

The technical background of reinforcement learning and deep Q-learning is studied and practically understood by implementing a representative solution to self driving car setup. It can be concluded that the reinforcement learning paradigm can be used for model based and model free environment, where Q-learning algorithm is used for solving model free reinforcement learning control problems.

References

[1] Ahmad El Sallab , Mohammed Abdou , Etienne Perot and Senthil Yogamani “Deep Reinforcement Learning Framework for Autonomous Driving”Autonomous Vehicles and Machines 2017

[2] 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”

[3] Richard Sutton and Andrew Barto, Reinforcement Learning: An Introduction (2nd Edition)

[4] http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

[5] https://github.com/harvitronix/reinforcement-learning-car