Cairo University
Faculty of Engineering
Electronics & Electrical Communication
Engineering Department

Lane Keeping Assist using Reinforcement Learning

A Graduation Project Report
Submitted in partial fulfillment of the requirements of the degree of
Bachelor of Science in Electronics & Electrical Communication Engineering

Submitted by

Ahmed Raafat Abdel-Wahab

Ahmed Ramzy El Qotb

Eslam Hussein Hassan

Omar Mohamed Kamal

Supervised by

Prof. Dr. Hanan Ahmed Kamal

Acknowledgements

This dissertation does not only hold the results of the year’s work, but also reflects the relationships with many generous and stimulating people. This is the time where we have the opportunity to present our appreciation to all of them.

First, to our advisor, Prof.Dr.Hanan Ahmed Kamal, we would like to express our sincere gratitude and appreciation for your excellent guidance, caring, patience, and immense help in planning and executing this project in a timely manner. Her great personality and creativity provided us with an excellent atmosphere for work, while her technical insight and experience helped us a lot in our research. Their support at the time of crisis will always be remembered.

Of course, we will never find words enough to express the gratitude and appreciation
that we owe to our families. Their tender love and support have always been the cementing force for building what we achieved. The all-round support rendered by them provided the much needed stimulant to sail through the phases of stress and strain.

Finally, we would also like to thank Eng.Mohammed Abdou Tolba who is Algorithms Software Engineer and Deep Learning Researcher at Valeo, who was always there for any questions, providing precious advice and sharing his time and experience.

Abstract

Lane keeping assist (LKA) is surely one of the most important features for self-driving cars. LKA can be defined as the vehicle keep moving in its lane with a constant speed and controlling only the steering angle. In this work, this feature has been achieved using Reinforcement Learning, which is one of the Machine Learning categories.

Reinforcement Learning is considered as a strong Artificial Intelligence paradigm which can teach machines through the environment interaction and learning from their mistakes. Despite its perceived utility, it has not yet been successfully applied in automotive applications. It has two categories: Discrete Action Algorithms and Continuous Action Algorithms. In this work, we have used Q-Learning which is a Discrete Action Algorithm and Deep Deterministic Policy Gradient (DDPG) which is a Continuous Action Algorithm. Both algorithms have been discussed in details from the very basics in the review section.

For any Machine Learning problem, we have two main phases: Learning phase and testing phase. During the Learning phase, the vehicle learnt moving between the two-lane markings from the interaction with the surrounded environment, so this is called the exploration phase. The vehicle then built the sufficient model in this phase. During the testing phase, the vehicle is tested to move on a new track to validate the built model. We have used Simulators for training instead of real vehicle. TORCS is an open source game with the Simulated Car Racing plugin that can provide us with the sufficient sensor readings needed to apply the algorithms.

We applied both Q-Learning and DDPG algorithms on the TORCS simulator, using the same training and testing lanes, and we proved that DDPG is a lot better algorithm than Q-Learning, as it has much more smooth and accurate performance. Also, in the DDPG the vehicle was faster, took less time in training and less number of episodes than Q-Learning to create the model, we proved that DDPG is more suitable for implementing our feature and more suitable for the enhancements in the future.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

LKA	Lane Keeping Assist
AI	Artificial Intelligence
DDPG	Deep Deterministic Policy Gradient
TORCS	The Open Racing Car Simulator
SCR	Simulated Car Racing
RL	Reinforcement Learning
MDP	Markov Decision Process
MC	Monte-Carlo
TD	Temporal Difference
SGD	Stochastic Gradient Decent
DQN	Deep Q-Network
UDP	User Datagram Protocol

Chapter 1
Introduction

This work aims to explore how Reinforcement Learning can be used to efficiently train a simulated car to drive autonomously and stay in the middle of the lane.

Reinforcement Learning is used in many applications, but mostly games because some of them simulates the real world perfectly, and after training on this game, we can then apply it on real life.

We have used the framework of the Simulated Car Racing (SCR) which is a plugin for The Open Racing Car Simulator (TORCS) instead of training on a real car, and this is because in training phase the car tries various random actions and therefore, it will surely get damaged.

At first, we will give a brief historical overview on artificial intelligence and it's relation with games and human, then we will define our problem and objective.

1.1. History

1952 — Arthur Samuel wrote the first computer learning program. The program was the game of checkers, and the IBM computer improved at the game the more it played, studying which moves made up winning strategies and incorporating those moves into its program.

1997 — IBM's Deep Blue beats the world champion at chess.

2006 — Geoffrey Hinton coins the term "Deep learning" to explain new algorithms that let computers "see" and distinguish objects and text in images and videos.

2011 — IBM's Watson beats its human competitors at Jeopardy.

2016__ AlphaGo versus Lee Sedol, or Google DeepMind Challenge Match, was a five-game Go match between 18-time world champion Lee Sedol and AlphaGo and

AlphaGo won 4 games to 1 then they made AlphaGo Zero who beat AlphaGo 100 games to 0.

This shows that there are great improvements in the Reinforcement Learning field, as a matter of fact it is still a research area and it is believed to be the future. In the following section we will define the problem and the objective of this work.

1.2. Problem Definition

The main problem here is to keep the vehicle driving at constant speed autonomously while keeping the vehicle in the middle of the lane, but the challenge is obvious here at the sharp turns of the lane, which makes the vehicle go out of it. The Reinforcement Learning makes a model and learns from its own experiences by taking actions. Additionally, we can convert Reinforcement Learning into a Supervised Learning problem and reduce its cost function.

In Machine Learning, we have two phases, the learning phase and testing phase. During the learning phase, we will use a simulated vehicle and train it on more than one track to generalize and learn how to drive by taking actions and getting the feedback of those actions (Rewards), we will choose tracks that has straight lines, left and right curves and sharp turns too. Of course, we can’t train on a real vehicle, as this would damage it completely. Then, we will go on to the testing phase, where we will test our simulated vehicle on a totally unknown track to test if it has learnt or not.

1.3. Objective

The proposed work aiming to have an autonomous driving vehicle with a Lane Keeping Assist feature using Reinforcement Learning field. This field is an interesting category of Machine Learning.

We have many algorithms in the Reinforcement Learning field: Discrete Action Algorithms, and Continuous Action Algorithms. Our aim is to implement both algorithms using Q-Learning as Discrete Action Algorithm and DDPG as continues Action Algorithm and compare the performances of both algorithms in order to know which one is better for our feature implementation (LKA). Our comparsion is based on the vehicle motion on track and keeping it in the middle of the track with an acceptable speed.

We will discuss the basics fundamentals of each algorithm used and how to implement it on TORCS step by step, showing which algorithm is better to solve our problem efficiently.

TORCS will help us with a simulated vehicle instead of using a real one. It will give us all the sensors needed as if it is a real vehicle in a real lane, and we will use these sensors and preprocess it, to help us guide our simulated vehicle to drive autonomously in the middle of the lane, without diverting out of it.

1.4. Outline of the chapters

In this work we have six chapters including the introduction chapter. The following chapters are organized as follows:

Chapter 2 is an overview on the different Machine Learning algorithms excluding Reinforcement Learning which will be our main topic. We will discuss the Supervised Learning and how it is done. Also discuss some Unsupervised Learning techniques.

Chapter 3 is a detailed review on the Reinforcement Learning techniques used in this work. We will first define what is Reinforcement Learning (RL), and how we model our problem using Markov Decision Process (MDP) framework, then go through the Model free and Model based approaches for solving the MDP and explain the difference between them, then we will explain what is Deep Deterministic Policy Gradient (DDPG).

Chapter 4 describes TORCS Simulator (TORCS) which allows us to implement our algorithms based on the provided sensors. These sensors provide us with the sufficient information about the surrounded environment. After that, it illustrates how we can consider TORCS as an MDP through its states and actions with the help of the provided sensors. Also, we describe how we applied Q-Learning on TORCS step by step through describing the game loop on TORCS and the technique of client-server and how we take actions from the sent states then added the results of the applied algorithm.

Chapter 5 describes how we applied DDPG on TORCS step by step through describing the game loop on TORCS and the technique of client-server and how we take actions from the sent states. And then provide the results of DDPG followed by comparison between both algorithms.

Chapter 6 illustrates conclusion of our work and the future works for Autonomous driving and Lane Keeping Assist.

Chapter 2
Machine Learning Algorithms

In this chapter we will illustrate Machine learning algorithms in brief, including Supervised Learning such as Linear Regression in Section 2.1, Unsupervised Learning such as clustering with some of its techniques in Section 2.2, then we will give very brief introduction about how Recommender systems work in Section 2.3.

2.1. Supervised Learning

It is the machine learning task of predicting a function from labeled training data.
the goal is to learn a general rule that maps inputs to outputs. It has 2 important algorithms: Linear Regression and Logistic Regression.

2.1.1. Linear Regression

Linear regression requires the dependent variable to be continuous i.e. numeric values (no categories or groups) and this can be clarified in Figure 1.1.

General equation of linear regression with one variable is: $H_\Theta(X) = \Theta_0 + \Theta_1.X$.
Idea: choose Θ_0, Θ_1 so that $H_\Theta(X)$ is close to Y for our training examples (X, Y).
To minimize the error we define a new function $J(\Theta_0, \Theta_1)$ which is equal to the square difference between Y and $H_\Theta(X)$.

$$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (H_\Theta(X^{(i)}) - Y^{(i)})^2$$

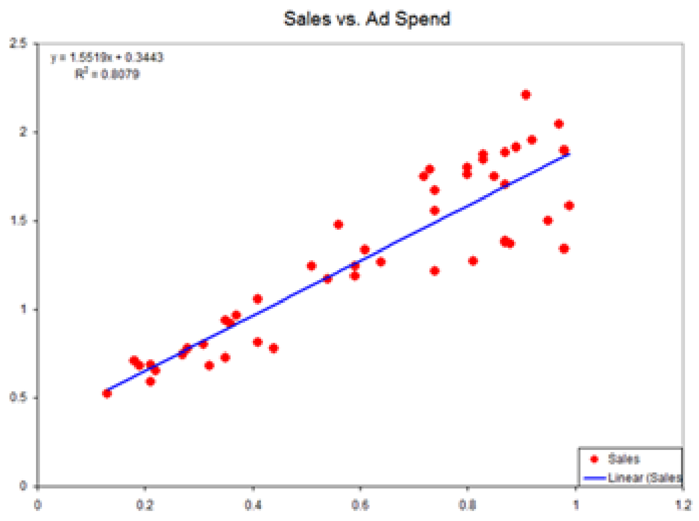


Figure 2-1 Linear Regression

Our goal now is to minimize the squared error function and this is done by using Gradient descent which keeps changing the parameters iteratively until it minimizes the error function.

Gradient descent algorithm

Repeat until convergence {

$\Theta_{j:} = \Theta_j - \alpha \frac{\partial y}{\partial j} J\ (\Theta_{0,} \Theta_1)$ (for j=0 and j=1)

}

Simultaneously update Θ_0 and $\Theta_{1.}$

Where: α is the learning rate, which controls how big a step we take with gradient descent. And from its name, it shows how fast or slow we want our model to learn, but we should also put into consideration that we might fall in local optimum and get stuck, which is not preferable. But as a solution for this, we decrease the learning rate.

In a nutshell

If α is too small, gradient descent can being slow but can guaranteed reaching the minimum.
If α is too large, gradient descent can overshoot the minimum.

Now we will explain the other supervised learning algorithm called Logistic Regression

2.1.2. Logistic Regression

It is the appropriate regression analysis to conduct when the dependent variable is binary it used mainly in classification problems. As shown in Figure 2.2.

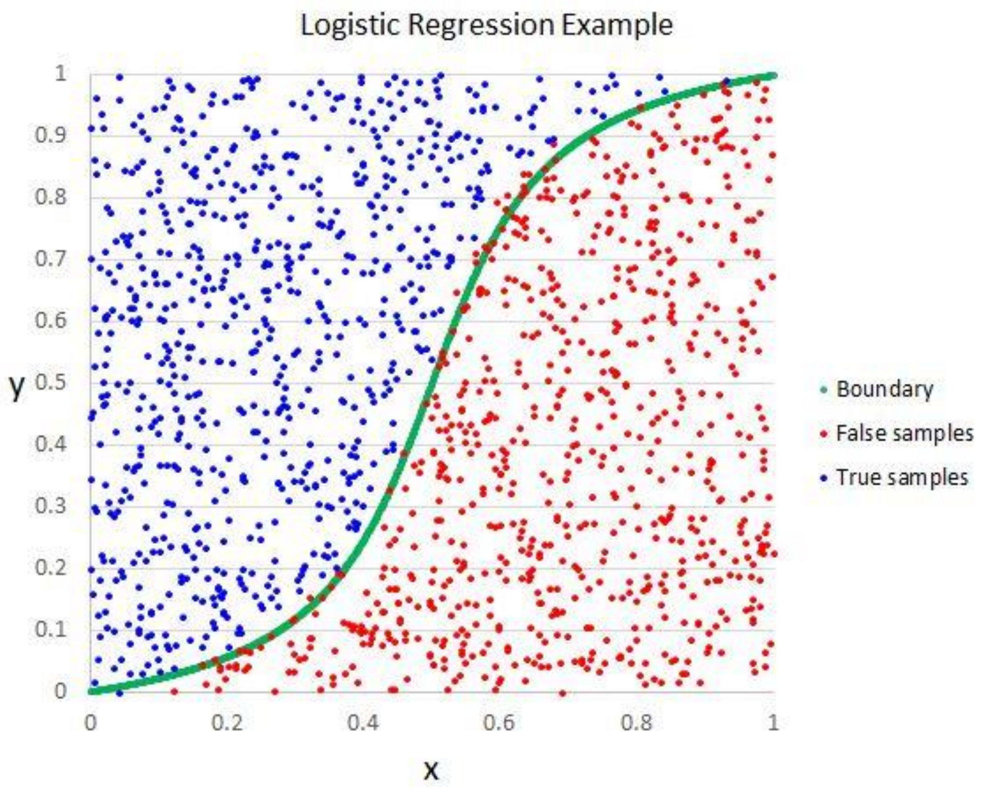


Figure 2-2 Logistic Regression example

Classification problems for logistic regression

Output is can be Binary (0 or 1) this called Binary Classification.

Or can be discrete Values and this called Multiclass Classification.

We will only discuss the Binary classification, as it is simpler.

- **Binary Classification**

Output Y ϵ {0, 1}

Where 0: negative class 1: Positive class

Logistic regression: $0 \leq H_\Theta(X) \leq 1$

Where Logistic regression hypothesis is the sigmoid function as follows:

$H_\Theta(X) = g(\Theta^T.X)$

$g(z) = \frac{1}{1+e^{-z}}$ (Sigmoid function)

sigmoid function is shown in Figure 2.3

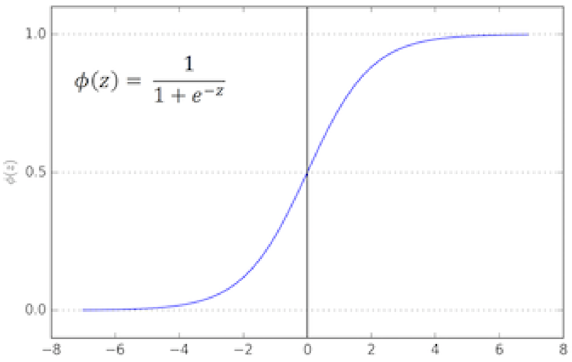


Figure 2-3 Sigmoid Function

predict "Y=1" if $H_\Theta(X) \geq 0.5$.
predict "Y=0" if $H_\Theta(X) \leq 0.5$.
also, the idea is to choose the parameters which make a good boundary for the given training sets.
A Cost function is used to fit a logistic regression model and is defined as follows:

Logistic regression cost function

$J(\Theta) = \frac{1}{m}\sum_{i=0}^{m} Cost(H_\Theta(X^{(i)}), y^{(i)})$
$Cost(H_\Theta(X\quad), Y) = -Y \log(H_\Theta(X)) - (1-Y) \log (1- H_\Theta(X))$,Y = 0 or 1 always
Cost function represents the penalty of the learning algorithm.
If the cost value is high, this means that the difference between the predicted value and the actual output is high, which means bad prediction.
And if the difference between them is low, this means that the predicted value is near to be the same of the actual output, which means good prediction.

This cost function can also be minimized using gradient descent as in linear regression.

Gradient descent is used to get $\min_{\Theta} J(\Theta)$

Repeat {

$\Theta_{j:} = \Theta_j - \alpha \frac{\partial y}{\partial j} J(\Theta)$

}

Simultaneously update all Θ_j

In case there are no labels we go to the unsupervised learning which groups similar datasets which will be discussed in the following Section 2.2.

2.2. Unsupervised Learning

No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning). There are 3 approaches to solve an unsupervised learning problem.

Approaches to unsupervised learning include

1. Clustering
2. Anomaly detection
3. Neural network

2.2.1. Clustering

It is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields.

Methods of clustering

- **K-means**: partitions data into k distinct clusters based on distance to the centroid of a cluster
- **Gaussian mixture models**: models clusters as a mixture of multivariate normal density components
- **Hierarchical clustering**: builds a multilevel hierarchy of clusters by creating a cluster tree

Applications of clustering

- Market segmentation
- Social network analysis
- Organize computing clusters
- Astronomical data analysis

2.2.2. Anomaly detection

Anomaly detection is a technique used to identify unusual patterns that do not conform to expected behavior. And we have 3 types as follows:

- **Point anomalies:** A single instance of data is anomalous if it's too far off from the rest. Business use case: Detecting credit card fraud based on "amount spent."
- **Contextual anomalies**: The abnormality is context specific. This type of anomaly is common in time-series data. Business use case: Spending $100 on food every day during the holiday season is normal but may be odd otherwise.
- **Collective anomalies**: A set of data instances collectively helps in detecting anomalies. Business use case: Someone is trying to copy data form a remote machine to a local host unexpectedly, an anomaly that would be flagged as a potential cyber-attack.

Applications

Intrusion detection, fraud detection, fault detection, system health monitoring, event detection in sensor networks, and detecting Eco-system disturbances. It is often used in preprocessing to remove anomalous data from the dataset in supervised learning.

Lastly, we will talk about Neural network which is commonly used nowadays for any unsupervised learning problem. And which will also be used in our work.

2.2.3. Neural network

Used in non-linear classification problems to find the perfect boundary for clustering the datasets and it depends mainly on the regularized logistic function, but the hypothesis function $H_\theta(X)$ changes to the sigmoid activation function $H_{\boldsymbol{\theta}}(X)=\frac{1}{1+e^{-\boldsymbol{\theta}T.X}}$ Where $\theta^{\mathbf{T}}$ is the transpose of the parameter vector.

It consists of input layer, hidden layer(s) and output layer as shown in Figure 1.4.

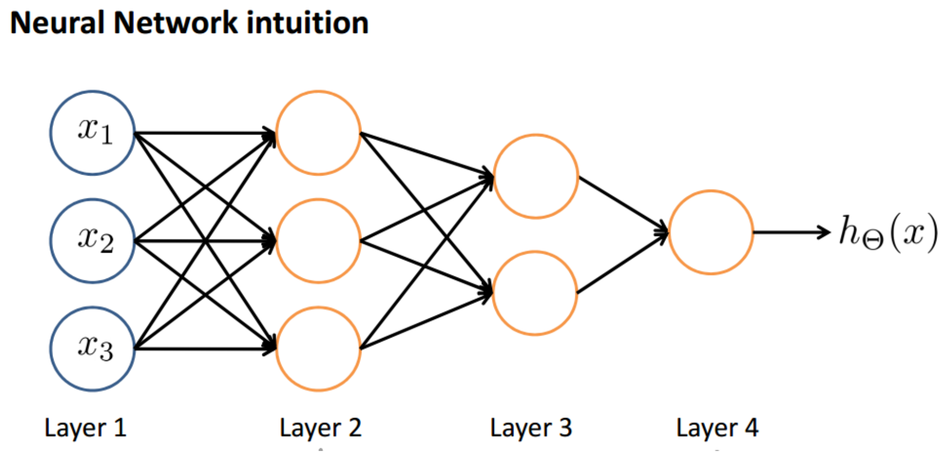


Figure 2-4 Neural Network example

Neural networks depends on two main categories: Forward propagation and Backward Propagation. They are essential for any network to learn and update its weights (Parameters) to eventually do our required function efficiently.

1- **Forward propagation:** Where you move from a layer to the next layer by taking the sigmoid of the multiplication of the activation layer by the weights (parameters) till you reach the output layer then you implement the backward propagation
2- **Backward propagation:** Where you compute the error between the output layer from the forward propagation and the expected output then by a mathematical trick we get the derivative of the cost error function which we want to minimize

Applications

Neural network has many applications in lots of fields.

- Image and character recognition
- Stock market prediction
- Medical diagnosis
- Process modeling and control
- Credit rating
- Speech recognition
- Fraud detection
- Financial forecasting
- Intelligent searching

2.3. Conclusion

Machine learning can be modeled using supervised and unsupervised learning techniques which can solve problems by finding a fitting curve and classification of a provided labeled data.

In the following Chapter 3 we will describe another machine learning algorithm called Reinforcement Learning which doesn’t deal with labels but can learn by interacting with environment by receiving feedback. This algorithm doesn’t need any supervision and uses reward approach.

Chapter 3
Reinforcement Learning

This part is an overview of the field of reinforcement learning and concepts that are relevant to the proposed work. The field of reinforcement learning is not very well-known and although the learning paradigm is easily understandable, some of the more detailed concepts can be difficult to grasp. Accordingly, reinforcement learning is presented at the beginning with a review of the fundamental concepts and methods for both discrete action algorithms and continuous action algorithms starting from the very basics.

This introduction to reinforcement learning is followed by a review of the three major components of the reinforcement learning method: the environment, the learning algorithm, and the representation of the learned knowledge. This representation is called Markov Decision Process (MDP) and it will be explained in this chapter in details.

3.1. Reinforcement Learning Definition

Reinforcement Learning is derived from training animals in Biology especially from a famous experiment done by Skinner in 1932 in Biology. Skinner trained a caged rat on pressing on a bar because of giving it a small food as a reward every time it pressed the bar. We can conclude from this experiment that we learned the rat to take an action based on the interacting with the environment which is represented in the cage and the bar.

Reinforcement learning is learning by interacting with an environment. An agent learns from its actions, and it selects its actions based on its past experiences (exploitation) and also by new choices (exploration), which is simply a trial and error learning. The agent receives a numerical reward, and the agent seeks to learn selecting actions that maximize the accumulated reward over time. But in reinforcement learning there isn’t a teacher that tells the agent what is the best action that must done, the agent must learn the effect of its actions in each state by trying them and observing the reward.

The following Figure 3.1 will illustrate what is reinforcement learning from another view.

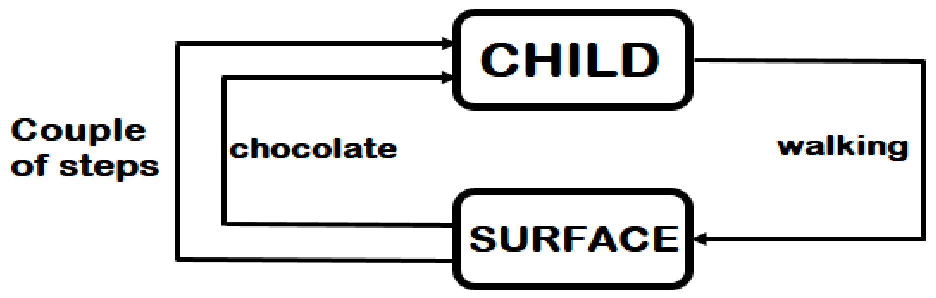


Figure 3-1 RL Child example

Consider an example of a child learning to walk.

Here are the steps of a child taking steps to learn walking:

1. The first thing the child will observe, is to notice how you are walking. You use two legs, taking a step at a time in order to walk. Grasping this concept, the child tries to replicate you.
2. But soon he/she will understand that before walking, the child has to stand up! This is a challenge that comes along while trying to walk. So now the child attempts to get up, staggering and slipping but still determinant to get up.
3. Then there's another challenge to cope up with. Standing up was easy, but to remain still is another task altogether! Clutching thin air to find support, the child manages to stay standing.
4. Now the real task for the child is to start walking. But it's easy to say than actually do it. There are so many things to keep in mind, like balancing the body weight, deciding which foot to put next and where to put it.

Let's formalize the above example, the "problem statement" of the example is to walk, where the child is the agent trying to manipulate the environment (which is the surface on which it walks) by taking actions (walking) and he/she tries to go from one state (each step he/she takes) to another.

The child gets a reward (let’s say chocolate) when he/she accomplishes a sub-module of the task (taking couple of steps) and will not receive any chocolate (negative reward) when he/she is not able to walk.

The following Figure 3.2 is a simplified description of a reinforcement learning problem.

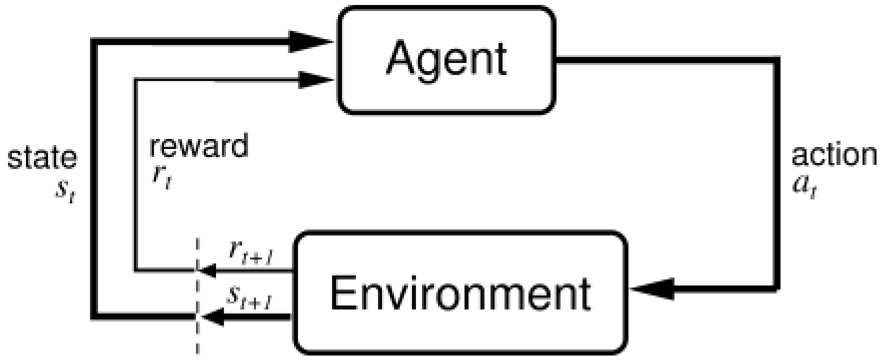


Figure 3-2 RL block diagram

In the next section we will represent RL problems generally with an environment, agent, state, reward and action as in Figure 3.2. And this will lead us to Markov Decision Process (MDP).

3.2. Markov Decision Process (MDP)

Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.

Formally, RL can be described as a Markov decision process (MDP), which consists of:

- A set of states S.
- A set of actions A.
- Transition dynamics $p(s_{t+1}|s_t, a_t)$ that map a state- action pair at time t onto a distribution of states at time t + 1.
- An immediate/instantaneous reward function R.
- A discount factor $\gamma \in [0, 1]$, where lower values place more emphasis on immediate rewards.

Policy is a solution for MDP which specifies an action for every state of the agent. And optimal Policy achieves the highest accumulated expected reward.
In general, the policy π is a mapping from states to a probability distribution over actions: π: S → p (A = a|S).

The following Figure 3.3 is an example for illustration the meaning of Policy.

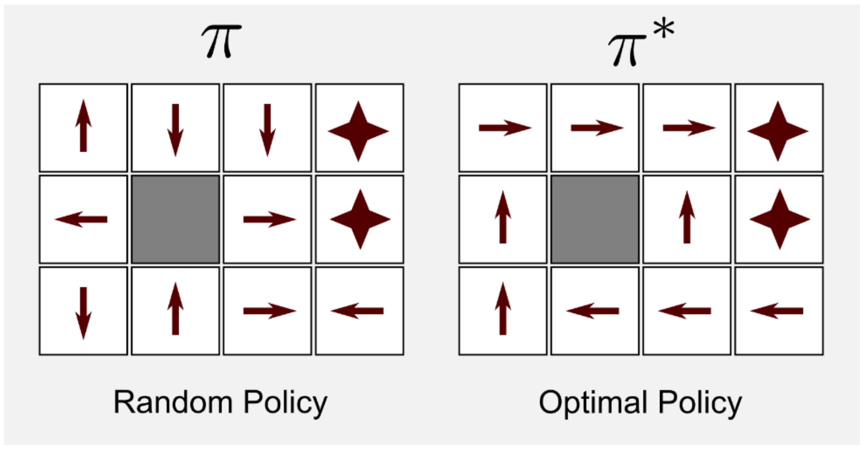


Figure 3-3 Random policy vs optimal policy

Let's consider in the previous example in Figure 3.3, that we have a robot and its target is reach to the first star, here we have 11 available blocks which represent 11 states, we have 4 actions at each state which are moving actions (right, left, forward, backward) and this is the difference between random policy and optimal policy.

3.3. Solution for MDP

The task of agent is to learn policy π: S→A for selecting its next action a_t based on the current observed state s_t that is, π(s) =a_t How shall we specify precisely which policy π we would like the agent to learn?

Bellman equation $(\boldsymbol{E_\pi[R_t + \gamma * v_\pi(S_{t+1})|S_t = s]})$ uses state and action value functions to find the optimal policy in a step by step fashion, where it:
1- Breaks the problem into small pieces
2- Launches iterative process until there is no small pieces remain
3- Reuses solved pieces to solve the next pieces

The maximum cumulative reward (return) is random due to the randomness of policy and could be also due to the randomness of environment which can have different reward and next state given current state and action and we can solve this by taking an expectation which corresponds to the Value function.

The solution of bellman equation means that we will solve the expectation which can be done by taking integration over all states and actions.

In the next Section we will define the Value and Action-Value function, which are used to solve the Bellman equation and find optimal policy. Specifically, solving the expectation from a statistical approach.

3.3.1. Value function (State-value function)

Depends only on the current state, it is the mean reward that an agent can get from environment and uses Bellman expectation equation:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{r,s\prime} p(r, s'|s, a)\ [r + \gamma * v_\pi(s')]$$

Where, $\sum_a \pi(a|s)$ is the policy stochasticity
$\sum_{r,s\prime} p(r, s'|s, a)$ is the environment stochasticity

With an approximation approach we can compute it as follows:

$$V_\pi(s) = R(s,a) + \gamma \cdot \sum_{t=0}^{\infty} p(r, s'|s, \mathrm{a}) \cdot V_\pi(s')$$

Then, value function at any state S is the expected accumulated following the policy from state S. but we want to deal with the optimal value so,

$$V^*(\mathrm{s}) = \max R(s,a) + \gamma \cdot \sum_{t=0}^{\infty} p(r, s'|s, \mathrm{a}) \cdot V_\pi(s')$$

3.3.2. Action-Value function

To ensure we have chosen the optimal action at every state, we need to define a function that calculates the value of taking an action in a certain state. This function is called action-value function (Q-function), the Q-value function at any state and action is the expected accumulator reward from taking an action in state S and then following the policy.

It isn't connected with the policy, which can have even have 0 probability.
Has no stochasticity in the first action step and we don't have to sum over possible initial actions

$$Q_\pi(s,a) = E_\pi[R_t + \gamma * v_\pi(S_{t+1})|S_t = s, A_t = a]$$

Solving the expectation:

$$Q_\pi(s,a) = \sum_{r,s\prime} p(r, s'|s, a)\ [r + \gamma * v_\pi(s')]$$

With approximation approach the optimal policy Q function can be defined as follows:

$$Q^*(\mathrm{s}) = R(s,a) + \gamma \cdot \sum_{t=0}^{\infty} p(r, s'|s, \mathrm{a}) \cdot V^*(s')$$

Then we can define the optimal policy as follows

$$\pi^*(s,a) = argmax(Q^*(s,a))$$

Therefore, we need something to solve this Value function and Action-Value function, in the next Section 3.4 we will deal with model based approach, which uses iterative methods for finding the value function, then manipulate the policy directly. And then in Section 3.5 we will look in a better approach for solving our problem which deals with model free approach which will appear to be computationally easier than model based.

3.4. Model Based approach

One approach that might immediately strike you, after framing the problem like this, is for the agent to learn a model of how the environment works from its observations and then plan a solution using that model. That is, if the agent is currently in state $s1$, takes action $a1$, and then observes the environment transition to state $s2$ with reward $r2$, that information can be used to improve its estimate of $P(s2|s1, a1)$ and $R(s1, a1)$, which can be performed using supervised learning approaches. Once the agent has adequately modeled the environment, it can use a planning algorithm with its learned model to find a policy.

Our target is to compute the optimal policy then we use dynamic programming algorithms to achieve that, the most well-known dynamic programming algorithms are:

1-Value iteration
2-Policy iteration

3.4.1. Value iteration

The first algorithm we will look at is value iteration. The basic idea of value iteration is if we knew the true value of each state, our decision would be always choose the action that maximizes expected utility. But we don’t initially know the state’s true value; we only know its immediate reward. But, for example, a state might have low initial reward but be on the path to a high-reward state.

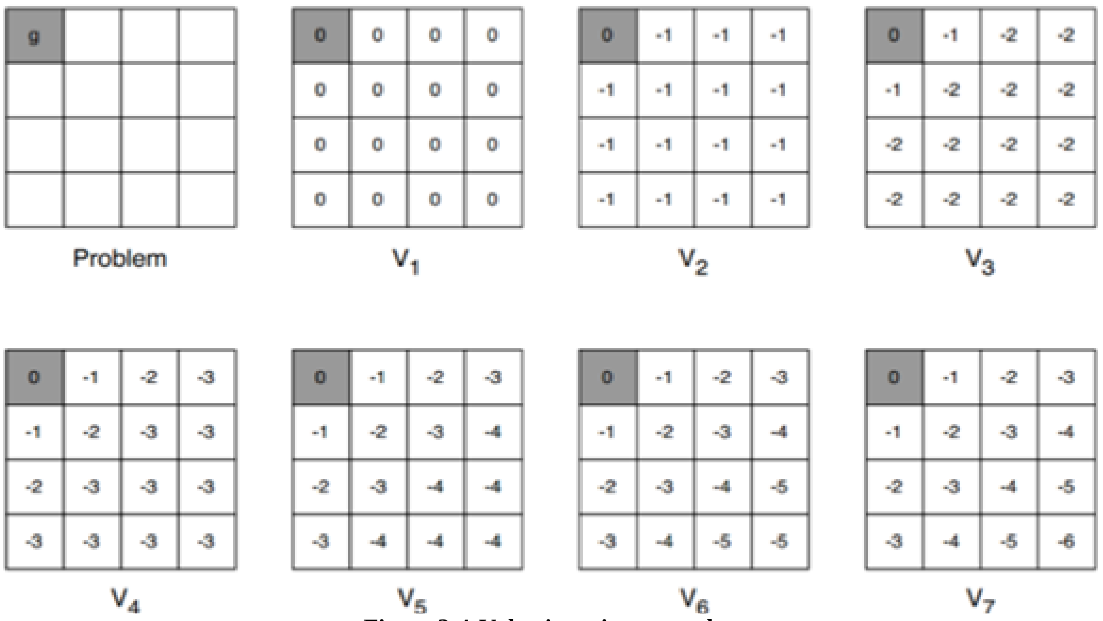
The true value of a state is the immediate reward for that state, plus the expected discounted reward if the agent acted optimally from that point on

$$V_{i+1}(\text{s}) \leftarrow \max \{\text{R(s, a)} + \gamma \textstyle\sum_{s'} P(s'|s,\text{a}) V_i(\text{s}')\}$$

Performs only 1 single policy evaluation and full policy improvement, starts with initializing v(s), then solves Bellman optimality equation $v(s') = \max_{\text{a}}(Q(s,a))$

Following these steps to solve the model in Figure 3.4:

1. Assign each state a random value
2. For each state, calculate its new V.
3. Update each state’s V based on the calculation above assuming that the single step has a reward of -1.
4. If no change in V after more iteration, halt. This algorithm is guaranteed to converge to the optimal solutions.
5. Repeat step 1 again.



Problem

g			

V_1

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_2

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_3

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_4

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_7

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

Figure 3-4 Value iteration example

3.4.2. Policy Iteration

Another way to get Optimal Policy is to use Policy Iteration, Policy Iteration algorithm manipulates the policy directly. In Policy Iteration algorithms, you start with a random policy, then find the value function for each state of that policy (policy evaluation step), then find a new (improved) policy based on the previous value function, and so on till get the optimal policy.
But, it requires precise policy evaluation before improvement step therefore we do many evaluations until numerical convergence of state values before doing single improvement

In this process, each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Given a policy, its value function can be obtained using the Bellman equation.

$$\pi'(s) \text{ <- } \mathrm{argmax}\,[R\,(s, a) + \gamma \textstyle\sum_{s'} p(s'|s,a) V_\pi\,(s')]$$

Example:

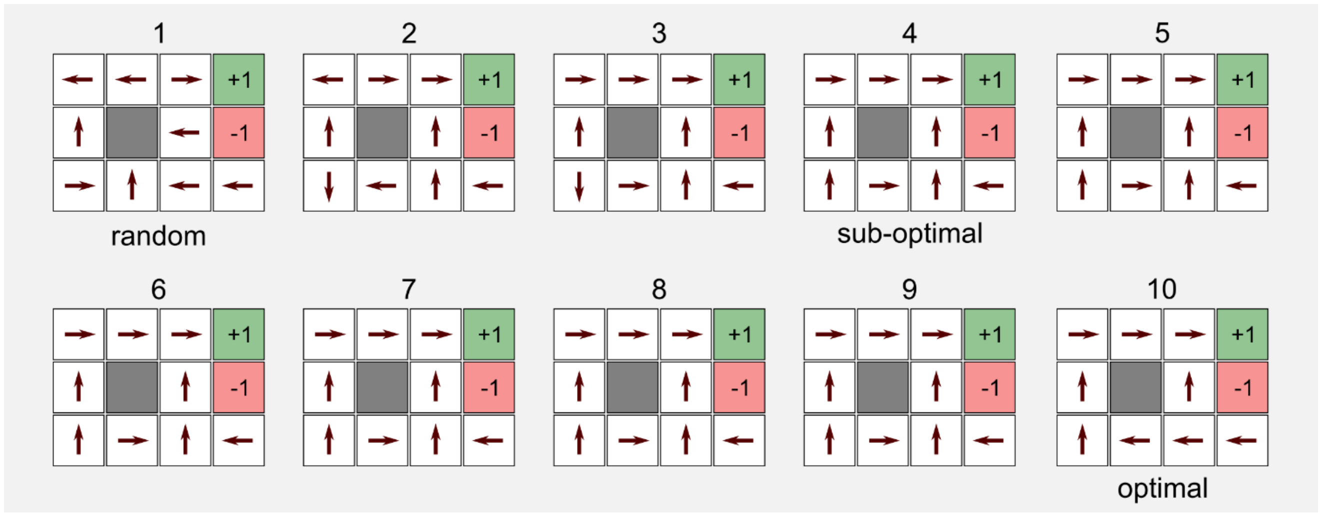


Figure 3-5 Policy iteration example

So, in policy iteration algorithm we make two steps:

Policy evaluation

Measures how good a given policy in each state

1- Initialize value function to zero or any initialization as long as terminal states are zero
2- Solve bellman expectation equation v(s) until state values do not change anymore
3- Gives us the direction where we use to improve our policy

$$v_\pi(s) = \sum_a \pi(a|s) \cdot Q(s,a)$$

Policy improvement

1- Recover Q from value function V which can be computed from policy evaluation
2- Find an action that maximizes Q function which is the best possible action
3- Based on Bellman optimality equation

If Q* is known, the optimal policy can be obtained by taking the action which maximizes its function.

$$\pi^*(s) = argmax(Q(s,a))$$

If V* is known, to recover the optimal policy from it, one need to recover first a Q function from this V*.

And this can be done with environment probabilities precisely in the same way as we did in policy improvement, and then recover this Q*.

Advantages of policy iteration

1- Doesn’t depend on initialization
2- Not susceptible to local optima
3- It doesn’t need complete policy evaluation
4- Doesn’t need to improve policy in all states at any particular step. As long as the policy is updated in each and every state once in a while.
5- Updating one state at each policy evaluation step will converge the GPI to global optima

Also updating it in random direction will converge to global optimal policy

3.5. Model-free approach (Discrete Action Algorithms)

We don't have information about transition probabilities or the reward function, as you don’t know anything about how the opponent can react, in other words you don’t know which state will come next.

You can sample from states and rewards from environment, but you don’t know the exact probability of them occurring. Therefore, we can’t compute the expectation of the possible outcome and this prevents using the optimal policy given value function. Because these approaches do not learn a model of the environment they are called model-free algorithms such as Q-Learning.

3.5.1. Q-Learning

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used in finding an optimal action-selection policy for any given (finite) MDP.

It comes from Q function that map states to action by computing the values of action in certain states instead of computing values of this states.

We only have access to the trajectories and not the whole possible states and actions transitions where the trajectory is a sequence of state, action and reward.

It can be computed using 2 different approaches:

1- Monte Carlo (MC)
2- Temporal difference (TD)

The following table 3-1 is showing the difference between MC and TD.

Table 3-1 Difference between Monte Carlo and Temporal Difference

Monte Carlo (MC)	Temporal Difference (TD)
- Needs full trajectory to learn - Average Q over sampled trajectories - Less reliant on markov property - Can be used in simple problems	- Learn from partial trajectory - Works with infinite MDP - Exploits Q values and finds the maximum of them - Needs less experience to learn

Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)

- Lower Variance
- Online learning
- Can learn using incomplete sequences

So, upon these advantages we will use the TD learning in our Q-learning algorithm.

Here is the equation which used to derive Q-function independent on the state values:

$$Q^{*}(\mathrm{s}) = \mathrm{R}\ (\mathrm{s},\ \mathrm{a}) + \gamma\ \sum_{s\prime} p(s'|s,a)\ max_{a}\ Q(s',a)$$

This leads to have the updated rule as follows:

$$Q(s,a)\ \text{<-}\ Q(s,a) + \alpha\ [\mathrm{R}\ (\mathrm{s},\ \mathrm{a}) + \gamma\ \ max_{a}\ Q(s',a)\ - Q(s,a)]$$

Where **α** is the learning rate.

The learning rate determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information. In fully deterministic environments, a learning rate of **α=1** is optimal.

It is obvious in the above equation that the algorithm depend on choosing an arbitrary value for Q at the beginning then take actions according to Q-table and then Update the Q-table by using the assumed value and do this process again (calculate the new Q Value and use it to update the Q-Table) till reach to optimal solution by iterative method and get the optimal policy.

Q-Learning algorithm

1- Initialize all Q-Values with zeros.
2- Loop: get samples <s,a,r,s'> from environment and compute the $\hat{Q}(s,a) = r(s,a) + \gamma \cdot \max_{a} Q(s', a)$.
3- Update using TD method $\hat{Q}(s,a) = \alpha \cdot \hat{Q}(s,a) + (1-\alpha) \cdot Q(s,a)$.

Problems with Q-learning

1- It fails miserably and falls in sub-optimal policy (finds the shortest path and leaves the safer path) as for cliff problem, the agent may fall in the cliff from the epsilon (random action) and the problem comes from taking maximum
2- To solve this problem we go to SARSA (s,a,r,s',a')

Comparison between model based and model free approaches:

Table 3-2 : Model based vs Model free

	Model based	Model free
Advantages	- Safe to plan exploration and can train from simulated experiences.	- Computationally less complex. - Needs no accurate representation of the environment in order to be effective. This makes them more fundamental than model-based methods.
Disadvantages	- Agent only as good as the model learnt. Also, sometimes this becomes a bottleneck, as the model becomes surprisingly tricky to learn. - Computationally more complex than model-free methods.	- Actual experiences need to be gathered in order for training, which makes exploration more dangerous. - Cannot carry an explicit plan of how environmental dynamics affects the system, especially in response to an action previously taken.

Due to Q-Learning has an unstable performance, therefore we will go through another algorithm which uses neural network to learn the optimal policy. As neural networks tend to show great performance, accuracy and stability referring to the DQN, Which has a replay memory buffer to sample trajectories to improve the learning process.

3.6. Deep Q-Network (DQN)

In order to transform an ordinary Q-Network into a DQN we will be making the following improvements:

1. Going from a single-layer network to a multi-layer network.
2. Implementing Experience Replay, which will allow our network to train itself using stored memories from its experience.
3. Utilizing a second “target” network, which we will use to compute target Q-values during our updates.

The 3 main additions of DQN over ordinary Q-Network described as following:

Addition 1: Convolutional Layers

An agent can learn to play video games, it has to be able to make sense of the game’s screen output in a way that is at least similar to how humans or other intelligent animals are able to. Instead of considering each pixel independently, convolutional layers allow us to consider regions of an image and maintain spatial relationships between the objects on the screen as we send information up to higher levels of the network. In this way, they act similarly to human receptive fields. Indeed, there is a body of research showing that convolutional neural network learns representations that are similar to those of the primate visual cortex. As such, they are ideal for the first few elements within our network.

Addition 2: Experience Replay

The second major addition to make DQNs work is Experience Replay. The basic idea is that by storing an agent’s experiences, and then randomly taking mini batches of them to train the network, we can more robustly learn to perform well in the task. By keeping the experiences, we prevent the network from only learning about what it is immediately doing in the environment and allow it to learn from past experiences. Each of these experiences are stored as a row of (state, action, reward, next state). The Experience Replay buffer stores a fixed number of recent memories, and as new ones come in, old ones are removed.

Addition 3: Separate Target Network

The third major addition to the DQN that makes it unique is the utilization of a second network during the training procedure. This second network is used to generate the target-Q values that will be used to compute the loss for every action during training. Why not just use one network for both estimations?

The issue is that at every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. In this way training can proceed in a more stable manner.

Instead of updating the target network periodically and all at once, we will be updating it frequently, but slowly. We found that it stabilized the training process.

As any algorithm, it has problems. But these problems are about stability which are as follows:

Instability problems

1- Sequential correlated data which may hurt convergence and performance. Solved using experience replay
2- Instability of data distribution due to policy changes. Solved using target network, where you use another network for target with different weight parameters, then update it frequently either hard update or soft update.
3- Unstable gradients due to the high variations of Q-values and unknown scale of rewards. Solved using reward clipping from -1 to 1 for less peaky Q-values.

When we use Q-Learning algorithm we found that the performance is oscillatory and we should have Continuous states and actions to improve the performance so we will introduce a review for continuous action algorithm and describe it because we will implement this algorithm later on TORCS.

3.7. Continuous Action Algorithms

Actually, the proposed work will deal with one of the main algorithms for the Continuous Action Algorithms which solves the MDP problem: Deep Deterministic Policy Gradient Algorithm (DDPG), but first, we want to explain policy gradients and give an example on Actor Critic Algorithm because the DDPG Algorithm is considered to be the extension of the Actor Critic Algorithm.

3.7.1. Policy gradient method

As we know we have 2 approaches for solving RL problem.

1- Value based

We learn state value v(s) or state-action value Q(s,a) and then infer policy given value function but you need perfect Q values for optimal policy.

2- Policy based

Explicitly learn probability or deterministic policy and they adjust them to maximize the expected reward.

So far, our policy has simply been to act greedily on some value-based function. What if we tried to learn the policy itself? We can represent this policy as the probability to take a certain action, given the state.

This brings with it some advantages and disadvantages. In the case of very high-dimension action spaces, it can take a lot of computing power to find the maximum value, and learning a policy allows us to bypass that step. We also get the benefit of stochasticity, or a random probability to choose one action compared to another. In some partially observed environments, random decision making is an essential piece of the optimal strategy. However, learning a policy often has higher variance and will take longer to train, and it may reach a local optimum.

We will first look at the objective of the policy gradient method and how to approximate it, to make it more convenient.

Objective of Policy gradient method

To compute the expected reward: $J = E[R(s, a, r, s')]$

The agent gets born in some random state, takes one action and observes the reward and then a new session begins. To solve this expectation, we should integrate over all possible states and actions because we have infinite continuous amount of options.

$$J = E[R(s, a, r, s')] = \int_s p(s) \int_a \pi_\theta(a|s)\, R(s, a)\, da\, ds$$

The first integral is the state visitation frequency (may depend on the policy if its complicated) but, the second integral is the probability of taking action in the state which maybe the table of all possible probabilities of all action probabilities.

Approximation of expected reward

We can approximate this expected reward to $J \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a \in z_i} R(s, a)$ where the first summation is the sampling of N sessions by following $\pi_\theta(a|s)$ and this is called **Monte-Carlo sampling**.

Computing gradient

But how can we compute the gradient $\frac{dJ}{d\theta}$ when theta isn't in the formula after approximation? Sol.: We can try some simple duct tape approach called Finite difference $\nabla J \approx \frac{J_{\theta+\varepsilon} - J_\theta}{\varepsilon}$, this will technically work but the policy changes slightly by the small value of epsilon.

More problems

1- This Finite difference technique requires lots of episodes for computing $J_{\theta+\varepsilon}$ and same number for computing J_{θ}.

2- We will suffer from the noise caused from sampling would be still much larger than the difference between the two policies especially if J is sufficiently small.

3- If you use large J, the gradients will be useless for anything more complicated than linear model.

Another method for computing gradients

By analogy from $\nabla \log\big(\pi(s)\big) = \frac{1}{\pi(s)} \cdot \nabla\pi(s) \;\; \rightarrow \;\; \pi(s) \cdot \nabla \log\big(\pi(s)\big) = \nabla\pi(s)$, we are going to apply it to our J to make it more convenient.

$J = \int_s p(s) \int_a \pi_\theta(a|s)\, R(s,a)\; da\; ds \;\; \rightarrow$
$\int_s p(s) \int_a \pi_\theta(a|s)\, \nabla \log\big(\pi_\theta(a|s)\big)\, R(s,a)\; da\; ds$

The second formula allows us to approximate by sampling over states and actions and it is called the log-derivative trick.

Advantage function

This function computes how good your algorithm is doing, which is the difference between Q function and value function $A(s,a) = Q(s,a) - V(s)$

So, we replace the Q with advantage function in our approximated policy gradient using baseline.

As we said before our continuous action algorithm DDPG is based mainly on Actor-Critic Algorithm so we will explain briefly Actor-Critic Algorithm.

3.8.2. Actor-Critic Algorithm

The AC model has two aptly named components: an actor and a critic. The actor takes in the current environment state and determines the best action to take from there. The critic plays the "evaluation" role by taking in the environment state and an action and returning a score that represents how good the action is for the state. As shown in figure 3.6 the actor-critic model.

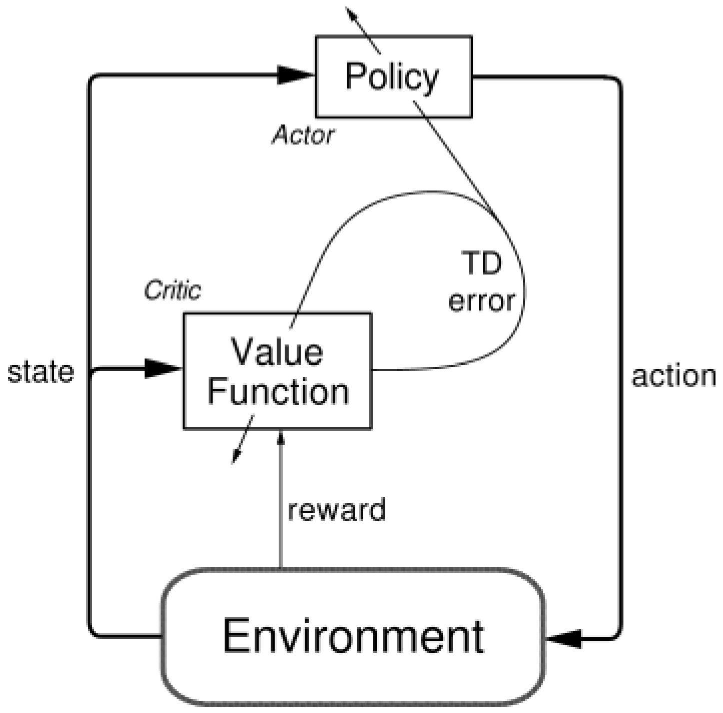


Figure 3-6 Actor-Critic Model

Imagine this as a playground with a kid (the "actor") and her parent (the "critic"). The kid is looking around, exploring all the possible options in this environment, such as sliding up a slide, swinging on a swing, and pulling grass from the ground. The parent will look at the kid, and either criticize or complement here based on what he did, taking the environment into account. The fact that the parent's decision is environmentally-dependent is both important and intuitive: after all, if the child tried to swing on the swing, it would deserve far less praise than if he tried to do so on a slide.

As shown in figure 3.7 the Actor-Critic represent the intersection between the value based and the policy based where the actor represent the policy based and the critic represent the value based.

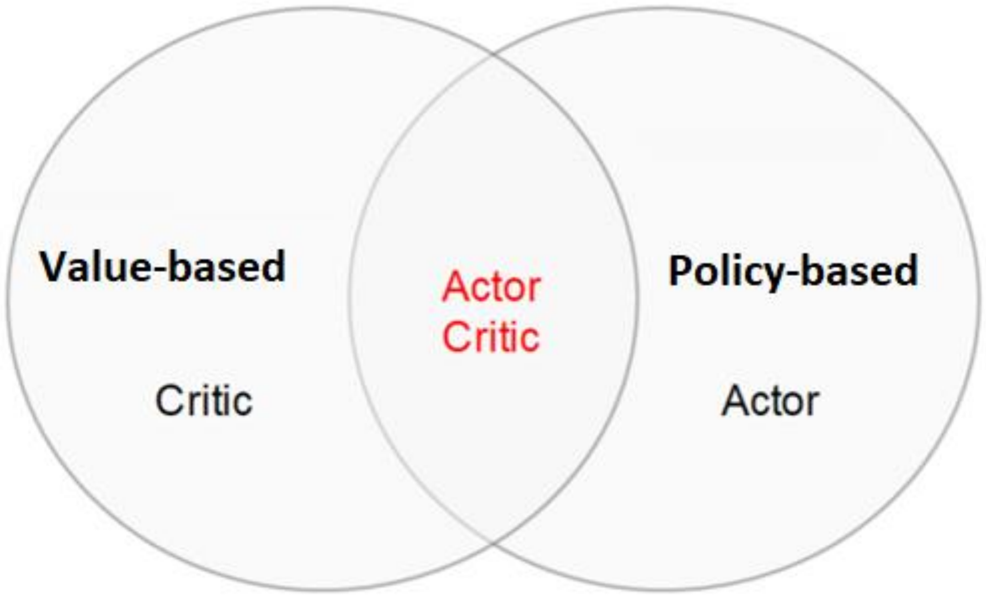


Figure 3-7 Actor-Critic

The critic part has a value based in which it tries to approximate the value function, and the actor Uses value function to learn policy function approximated same way.

Advantage Actor-Critic

Compute advantage function and replaces Q function in policy gradient method, so it becomes

$$\nabla J \approx \frac{1}{N}\sum_{i=0}^{N}\sum_{s,a\in z_i} \nabla \log \pi_\theta(a|s)\cdot A(s,a)$$

Where, $A(s,a) = r + \gamma \cdot V(s') - V(s)$, $Q(s,a) = r + \gamma \cdot V(s')$

Of course, we need to get expectation over all states for computing the Q function but we can approximate it by taking one sample as in Q learning.

Now, how to get the Value function?

1- Train a network that has two outputs, first it has to learn a policy with units of number of actions and then use SoftMax to learn the probability distribution and the second it estimates a value function which is a single neuron

2- Update the policy by using the V function to provide better estimate of policy gradient which is the equation above

You have to refine your value function by computing MSE and reducing the error and this way we can converge the expectation of value function.

$$L_{\text{critic}} \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a \in z_i} (V_\theta(s) - [r \cdot \gamma \cdot V(s')])^2$$

More on Advantage Actor-Critic

We have two losses. The first loss is the policy-based loss (policy gradient), the second one is you minimize temporal difference loss (value based).

We can consider the second loss to be less important. For a perfect critic but random actor, the critic will estimate how well the random agent perform, but if a good actor but random critic, actor still learns as good as other algorithms as Reinforce

The problem with actor critic that it is on-policy because you have to train it on the actions taken under its own policy and we can’t use experience replay.

So, if we want to use experience replay, we have to overcome the problem of having independent sessions with non-identical distribution. There is a study says if you run parallel sessions, then you can consider it ideal.

So, we can spawn multiple agents playing independent replicas of the environment with different weights, where they have to take actions by sampling from the policy independently. This way we make it off-policy.

3.8.3. Deep Deterministic Policy Gradient (DDPG)

This Algorithm is considered as the natural extension for the Actor Critic Algorithm by replacing the Actor and Critic parts with Actor and Critic Networks or models.

We will first explain the meaning of DDPG then explain what does it consist of and what are the inputs and outputs.

Deep

Instead of applying the normal Q-Learning algorithm, we will use Deep Q-Network in which we can replace the Q-Function by a neural network, $Q(s,a) \approx Q(s,a,w)$ where is the weights of the neural Network. And then transforming it to a supervised learning problem.

Deterministic

It means that for a given state, we have a particular action should be taken based on the learnt policy $a = \mu(s)$.

Policy Gradient

As we defined our reward function and went through derivations of solving the expected reward in Section 3.7.1.

DDPG is an actor-critic algorithm as well; it primarily uses two neural networks, one for the actor and one for the critic. These networks compute action predictions for the current state and generate a temporal-difference (TD) error signal each time step.

The input of the actor network is the current state, and the output is a single real value representing an action chosen from a continuous action space. The critic's output is simply the estimated Q-value of the current state and of the action given by the actor. The deterministic policy gradient theorem provides the update rule for the weights of the actor network. The critic network is updated from the gradients obtained from the TD error signal.

In TORCS, we used Policy Gradient method as the Actor model, and Q-Learning Algorithm with some modifications as the Critic model. We will explain the DDPG in this Section in brief. For more details, please refer to Q-Learning in Section 3.5.1. and Policy Gradient in Section 3.7.1.

3.9. Conclusion

We have discussed a detailed explanation of the Reinforcement learning technique and how we can solve it from a statistical approach and methods of approximation of the equations to make it more convenient to be used for implementation.

In the following Chapter , we will talk about the simulator used for implementing our algorithms, which is TORCS Simulator, then implement the reinforcement learning algorithms: Discrete Action Algorithm, specifically, Q-Learning algorithm.

Chapter 4
Q-Learning Implementation on TORCS

In this chapter we will discuss how to implement reinforcement learning algorithm in simulator instead of real car. So first we will talk about TORCS in Section 4.1., which is the simulator that's mentioned above then we will explain how to interface with it in Section 4.2. And how to run our algorithm on TORCS and process on its sensor and how to take actions. Finally, we will discuss our steps in coding, the loop game and the implementation of Q-learning algorithm on TORCS then show our results.

4.1. TORCS

The Open Racing Car Simulator as shown in Figure 4.1. TORCS is a modern, modular, highly-portable multi-player, multi-agent car simulator. Its high degree of modularity and portability render it ideal for artificial intelligence research.

Figure 4-1 Screenshot from TORCS

TORCS can be used to develop artificially intelligent (AI) agents for a variety of problems. At the car level, new simulation modules can be developed, which include intelligent control systems for various car components. At the driver level, a low-level Application program interface (API) gives detailed (but only partial) access to the simulation state. This could be used to develop anything from mid-level control systems to complex driving agents that find optimal racing lines, react successfully in unexpected situations and make good tactical race decisions.

Each on-going race is referred to as a simulation in TORCS and is described through many different data structures. The race situation is updated every 2 milliseconds (500 Hz), including updating the various mathematical models governing the physics of the race, e.g. motion and positioning of the cars and other objects.

4.2. SCR Championship Software

The Open Car Racing Simulator that has an architecture as shown in Figure 4.2, is a highly customizable open source car racing simulator that provides a sophisticated physics engine, 3D graphics, various game modes, and several diverse tracks and car models. Because of this, it has been used in the Simulated Car Racing championship since 2008

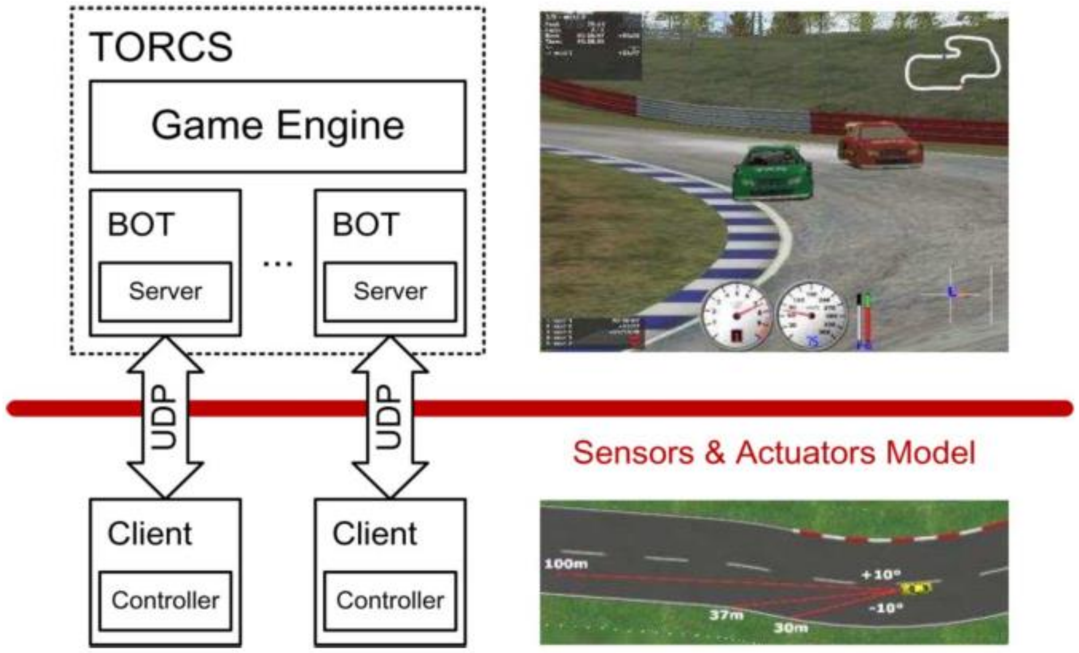


Figure 4-2 SCR architecture of TORCS

4.2.1 SCR Plug-in

Normally, the cars in TORCS have access to all information, including the environment and, to a certain degree, other cars. This is not representative of autonomous agents acting in the real world.

The server acts as a proxy for the environment and the client provides the control for a single car. The controllers run as external programs and communicate with a customized version of TORCS through UDP connections.

The server sends the client the available sensory input. In return, it receives the desired output of the actuators
This separates the controller from the environment, allowing it to be treated as an autonomous agent

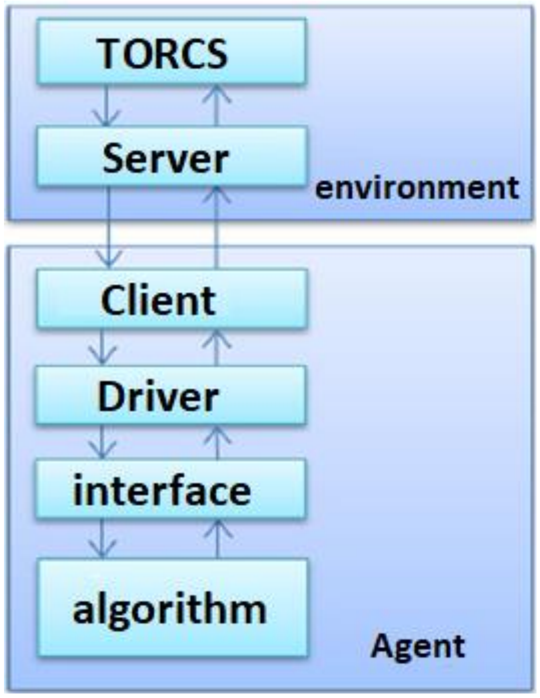


Figure 4-3 Block diagram of TORCS with RL

SCR Plugin is used to access the sensors which TORCS environment measure, and also give us the ability to send our action after being calculated from the algorithm and this is clear in Figure 4.3.

TORCS Sensors in Table 4.1 and Actions in figure 4.2 are as follows:

4.2.2. TORCS Sensors

Table 4-1 : Description of availble sensors in TORCS

Sensor	Description
Angle	Angle between the car direction and the direction of the track axis.
curLapTime	Time elapsed during current lap.
Damage	Current damage of the car (the higher is the value the higher is the damage).
distFromStartLine	Distance covered by the car from the beginning of the race.
Distracted	Distance of the car from the start line along the track line till the position of the car.
Fuel	Current fuel level.
Gear	Current gear: -1 is reverse 0 is neutral and the gear from 1 to 6.
lastLapTime	Time to complete last lap. Opponents: Vector of 36 sensors that detects the opponent distance in meters (range is [0,100]) within a specific 10 degrees sector: each sensor covers 10 degrees.
racePos	Position in the race with respect to other cars.
rpm	Number of rotations per minute of the car engine.
speedX	Speed of the car along the longitudinal axis of the car.
speedY	Speed of the car along the transverse axis of the car.
Track	Vector of 19 range finder sensors: each sensor represents the distance between the track edge and the car. Sensors are oriented every 10 degrees from –π/2 to +π/2 in front of the car. Distance are in meters within a range of 100 meters. When the car is outside of the track (i.e. track Pos is less than -1 or greater than 1), these values are not reliable!
trackPos	Distance between the car and the track axis. The value is normalized w.r.t. the track width: it is 0 when the car is on the axis, -1 when the car is on the left edge of the track and +1 when it is on the right edge of the car. Values greater than 1 or smaller than -1 means that the car is outside of the track.

wheelSpinVel	Vector of 4 sensors representing the rotation speed of the wheels.

4.2.3. TORCS Control Actions

Table 4-2 : Description of available actions in TORCS

Action	Description
Accel	Virtual gas pedal (0 means no gas, 1 full gas).
Brake	Virtual brake pedal (0 means no brake, 1 full brake).
Gear	Gear value
Steering	Steering value: -1 and +1 means respectively full left and right, that corresponds to an angle of 0.785398 rad.
Meta	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

4.3. TORCS as MDP

The model is developed under the Markov Decision Process (MDP) framework, which is a tuple of (S, A, P, γ, R) where:

a) S is set of Environment states

b) A is set of actions

c) R is reward

4.3.1. States

It’s clear that most of the sensor readings as shown in Figure 4.4 represent car states and it is very important for control the car. The states are the speed along the track, the position on the track, the angle with respect to the track axis and five distance sensors that measure the distance to the edge of the track. The 20◦ inputs are not taken directly from sensors 7 and 11, but computed as an average over sensors 6, 7, 8 and 10, 11, 12, respectively, to account for noise as shown in Table 4.3.

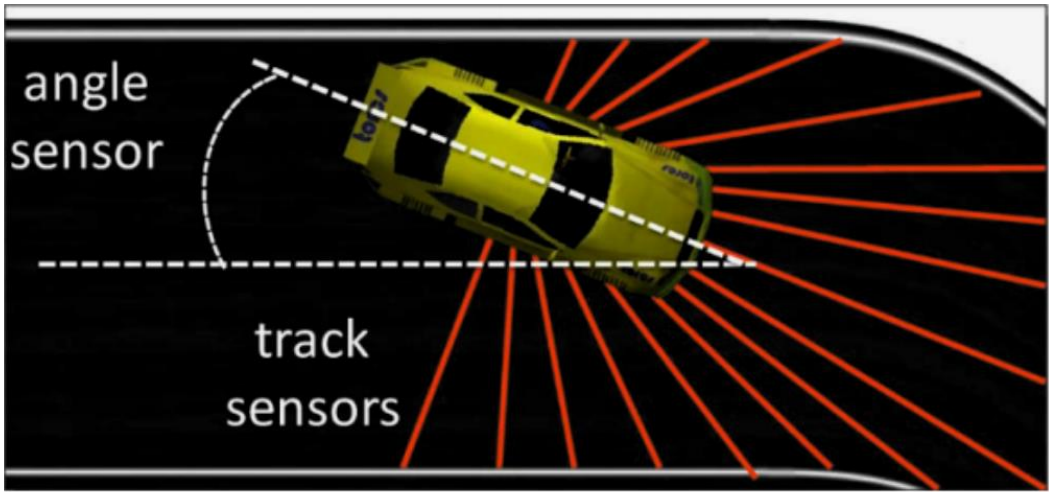


Figure 4-4 Distance sensors around the vehicle

Table 44-3 : Used TORCS States

Sensor name	State Description
Speed X	Speed of the car along the longitudinal axis of the car
angle	Angle between the car direction and the direction of the track axis.
Track pos	Distance between the car and the track axis
Distance sensor at −40◦, −20◦, 0◦, 20◦, 40◦	Distance between the car and track [5,avg(6, 7, 8), 9, avg(10,11,12), 13]

4.3.2. Actions

There are five action dimensions available in TORCS accelerate, brake, gear, meta and steer. Since braking is simply a negative acceleration, we shall view this as the negative side of the same dimension.

They are the basic controller of the SCR and the actions we will use will be explained in the following Table 4.4.

Table 4-4 : Used TORCS Actions

Name	**Range**	**Description**
accel	[0,1]	Virtual gas pedal (0 means no gas, 1 full gas).
brake	[0,1]	Virtual brake pedal (0 means no brake, 1 full brake)
gear	-1,0,1,2,3,4,5,6	Gear value.
steer	[-1,1]	Steering value: -1 and +1 means respectively full right and left
meta	0 or 1	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race

Accel

The values for acceleration were tuned manually as well. Since human players tend to balance between full acceleration and no acceleration at all and since the time resolution is quite high (one action per 20ms), it seemed unnecessary to give the agent a high resolution in the acceleration dimension. The agent was given three values: 1, 0 and -1,

Actually, steering correlated with acceleration and brake so the resulting discretization give us the following combinations in Table 4.5

Table 4-5 : TORCS Discrete Actions

Steer	Accelerate (1)	Neutral (0)	Brake (-1)
0.5 (left)	0	1	2
0.1 (left)	3	4	5
0	6	7	8
-0.1 (right)	9	10	11
-0.5 (right)	12	13	14

Gear

Shifts gear according to the rpm of the car's engine in Gear shifting values table as shown in Table 4.6.

Table 4-6 : Gear Shifting Values

Gear	1	2	3	4	5	6
Shift up	8000	9500	9500	9500	9500	0
Shift down	0	4000	6300	7000	7300	7300

Steer

We should make a discretization of these dimensions is needed. Initially the agent was given seven steering actions: -1, -0.5, -0.1, 0, 0.1, 0.5, and 1. This includes actions for small deviations and actions with a larger impact to pass sharp curves, but to provide sharp edges we remove (-1) and (1)

4.3.3. Rewards

Due to random actions there are good actions and bad actions, we want to achieve our target so we want to prevent the car to

1-Going out of the track
2-Stopping in a certain position
3-Making bad actions

To calculate the rewards, we have 3 situations:

1- Car make good lane keeping
    The reward will be positive and high if the distance was long and in general it has a continuous range from [-1, 1]
2-Stop in certain position
    The reward will be negative and equal to -1
3- The car goes out of the track
    The reward will be negative and equal to -1 and we will restart the race

In the previous Section 4.3, we have explained how to represent TORCS as an MDP, and how does it give us sufficient information as sensors. In the next Section 4.4, we will explain how the agent work by explaining the game loop as shown in Figure 4.5 at the end of the Section 4.4.

4.4. The Game Loop

Every 20 milliseconds, the SCR server asks the driver to return an action by calling its drive function. But this drive function depends on many other functions which are considered as the interface between the driver function and TORCS, and we will explain them and give an example.

Interfacing functions of driver function and TORCS.

4.4.1. Code contents

carControl
It has the control functions that control the car in the game, the driver put values for it then parses and sends it to the server to move the car with the given actions.
For example:
There is a function called **setSteer** which controlling the steer value of the car, the driver can put it with any value between -1 and 1 and then it to the server to apply it on the agent (car).

carState
It has all the sensor values which describe the state of the agent, distance from start, damage taken and other sensor readings like speed in all directions, track position and distance from 19 different angles.
For example:
getSpeedx is a function which holds the speed of the car in the x direction.

msgParser
It has a UDP message builder and receiver for the server-client communication. It builds the messages of the control actions and sends it to the server then receives the UDP messages from the server which is the car state.

Pyclient
It is the client code that connects to the server host given a specific port and socket and then calls the driver function which is our game loop and then uses the **msgParser** to transfer the UDP messages to the server.

We will first implement the driver function and check all the corner cases and conditions, then show the learning interface and how we choose action, then show a sample of the result and a sample of the Q-Table.
The driver function uses the previous functions to create the actions needed and send these actions to TORCS simulator to move the car. But to create these actions, we have to make sure that the car isn't stuck at any point and this can be done as follows:

Check Stuck

This function check that the car is stuck or not.
If the car's angle is larger than 45 degrees, it is considered stuck.
There are two steps to restart the game if the car is stuck:

- If it is stuck for more than 25 game ticks.
- The traveled distance is less than 0.01m.

In this case the episode ends and we start a new episode.
Every time the agent is not stuck, the stuck timer is reset to zero.

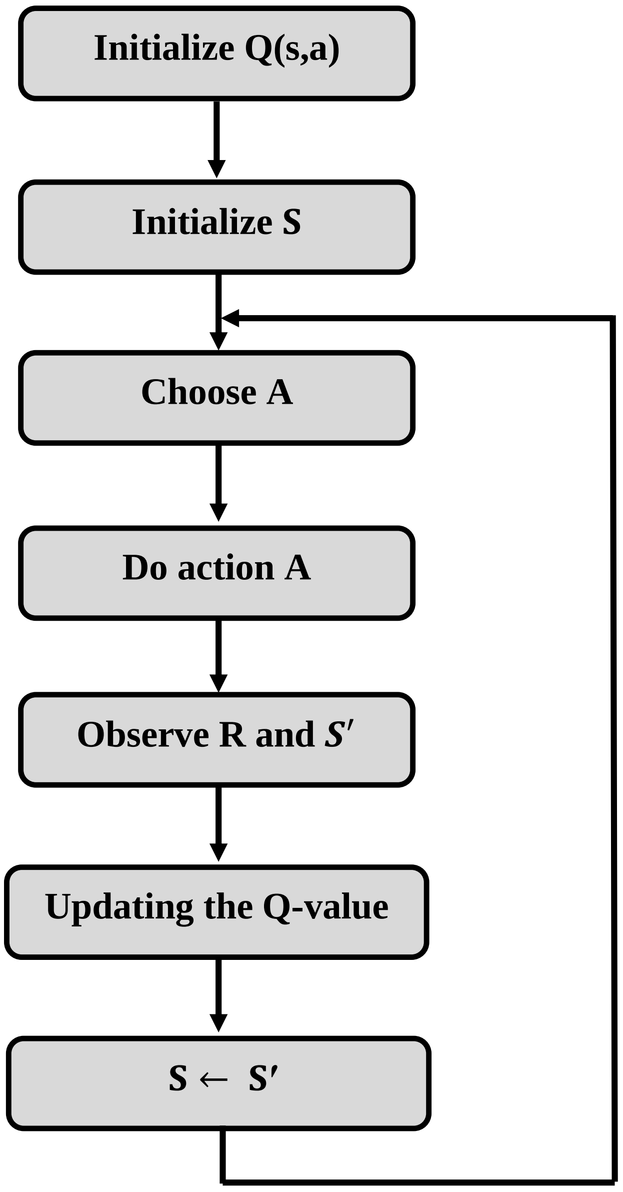
4.5. Q-Learning Interface

In this section we will explain the Q-Learning Algorithm, produce a flowchart for the algorithm and explain briefly state determination, action selection, the reward function and how to update the Q-Table.

4.5.1. Q-learning Algorithm

```
Initialize Q(s,a)
Repeat (for each episode):
    Initialize S
    Repeat (for each step in the episode):
        Choose A upon S using epsilon-greedy policy
        Do action A
        Observe reward (R) and the next state ($S'$)
        Updating the Q-value using the following equation:
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \cdot max_a Q(S', A) - Q(S, A)]$
        Set the next state to be the current state: $S \leftarrow S'$
    Until S is terminal
```

The following chart describe the Q-learning algorithm:



The primary function of the learning interface is:

- Do action selection
- Call the update function of the learning algorithm.

Learning interface consists of four functions:

- GetState which determine the state of the car.
- ActionSelection which choose the appropriate action in the determined state.
- RewardFunction which calculate the reward of the selected action.
- QtableUpdate which update the Q table by calculating the new Q value of the selected action.

GetState

First, we discretize the distance sensor and speed values as follows:

- **speedList** = [0,10,20,30,40,50,60,70,80,90,100,110,120,130,140,150]
- **distList** = [-1,0,5,10,20,30,40,50,60,70,80,90,100,120,150,200]

We chose these values carefully to cover all possible states.
We represented each of them in 4 bits of binary form as they are 16 discretized values.

To make the state of the car function of the sensors related to the target of the project which keep the car in determined lane we will use track sensors as a part of state description.

We have 19 track sensors we will take five readings only {sensor 5, average of (sensor 6, sensor 7, sensor 8), sensor 9, average of (sensor 11, sensor 12, sensor 13), sensor 1}.

To reduce the number of bits representing the state, we used only 1 sensor out of 5 sensors in which we take the maximum, as representing all the 5 sensors readings we will need to represent them in 20 bits, which will make the number of states equals to 2^{20} which is infeasible.

Therefore, we used another 3 bits to distinguish between the maximum of the 5 sensors which gives us of total 7 bits describing the sensor values.
Eventually, we have a total of 11 bits for the states which corresponds to 2048 possible state

E.g. Distance sensor = 200, Maximum sensor No. is 9, Speed = 90, this is equivalent to Figure 5.1 as shown.

Table 4-7 : Example on sensor discretization

Speed	Max. Sensor No.	Sensor value
1001	010	1111

Action Selection

We discretized the actions into 15 discrete values as we said in Table 4.5:
Action selection is based on a random number (ρ) between 0 and 1

$$\pi(\mathrm{s}) = f(x) = \begin{cases} heuristic\ action & if\ \rho < \eta \\ random\ action & if\ \rho < \eta + \epsilon \\ \max action & otherwise \end{cases}$$

Every time step there is a probability η of taking a heuristic action, a probability ϵ of taking a random action, and a probability 1 - η - ϵ of taking a greedy action (max action).

The heuristic action is more used as a guide than as a teacher.
Therefore, the random exploration is necessary to learn to improve upon the heuristic policy.
Max action is the action which has the highest Q-value in the Q-table given the current state.

Reward Function

We have 3 different scenarios:

If the car is stuck

It takes -2 reward and sends a META action to restart the episode, because it is an undesirable action, therefore we make sure it never happens again

If the car is out of track (abs (track position) > 1)

It takes -1 reward

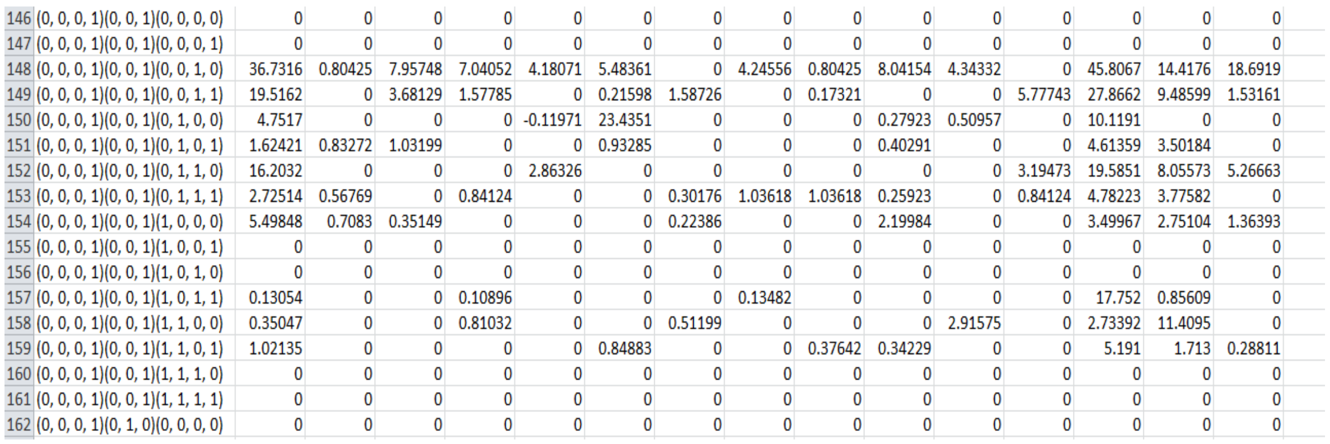
If the car is neither stuck or out of track

It takes a reward depending on track position, angle and travelled distance with a max value of 1

- **Q table Update**

The Figure 4.5 is a sample from the Q-Table which consists of 2048 states and 15 discretized actions, the actions were discussed in this Chapter in details.

First, we check if the current state already exists in the table, if not we create it in the



146	(0, 0, 0, 1)(0, 0, 1)(0, 0, 0, 0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
147	(0, 0, 0, 1)(0, 0, 1)(0, 0, 0, 1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
148	(0, 0, 0, 1)(0, 0, 1)(0, 0, 1, 0)	36.7316	0.80425	7.95748	7.04052	4.18071	5.48361	0	4.24556	0.80425	8.04154	4.34332	0	45.8067	14.4176	18.6919
149	(0, 0, 0, 1)(0, 0, 1)(0, 0, 1, 1)	19.5162	0	3.68129	1.57785	0	0.21598	1.58726	0	0.17321	0	0	5.77743	27.8662	9.48599	1.53161
150	(0, 0, 0, 1)(0, 0, 1)(0, 1, 0, 0)	4.7517	0	0	0	-0.11971	23.4351	0	0	0	0.27923	0.50957	0	10.1191	0	0
151	(0, 0, 0, 1)(0, 0, 1)(0, 1, 0, 1)	1.62421	0.83272	1.03199	0	0	0.93285	0	0	0	0.40291	0	0	4.61359	3.50184	0
152	(0, 0, 0, 1)(0, 0, 1)(0, 1, 1, 0)	16.2032	0	0	0	2.86326	0	0	0	0	0	0	3.19473	19.5851	8.05573	5.26663
153	(0, 0, 0, 1)(0, 0, 1)(0, 1, 1, 1)	2.72514	0.56769	0	0.84124	0	0	0.30176	1.03618	1.03618	0.25923	0	0.84124	4.78223	3.77582	0
154	(0, 0, 0, 1)(0, 0, 1)(1, 0, 0, 0)	5.49848	0.7083	0.35149	0	0	0	0.22386	0	0	2.19984	0	0	3.49967	2.75104	1.36393
155	(0, 0, 0, 1)(0, 0, 1)(1, 0, 0, 1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
156	(0, 0, 0, 1)(0, 0, 1)(1, 0, 1, 0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
157	(0, 0, 0, 1)(0, 0, 1)(1, 0, 1, 1)	0.13054	0	0	0.10896	0	0	0	0.13482	0	0	0	0	17.752	0.85609	0
158	(0, 0, 0, 1)(0, 0, 1)(1, 1, 0, 0)	0.35047	0	0	0.81032	0	0	0.51199	0	0	0	2.91575	0	2.73392	11.4095	0
159	(0, 0, 0, 1)(0, 0, 1)(1, 1, 0, 1)	1.02135	0	0	0	0	0.84883	0	0	0.37642	0.34229	0	0	5.191	1.713	0.28811
160	(0, 0, 0, 1)(0, 0, 1)(1, 1, 1, 0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
161	(0, 0, 0, 1)(0, 0, 1)(1, 1, 1, 1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
162	(0, 0, 0, 1)(0, 1, 0)(0, 0, 0, 0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-5 Q-Table sample

Q-table initialized with 0 for all actions.

Then, using the current state, the previous state, action and reward we update the Q-values of the previous state using the following update equation.

$$Q(s,a) \text{ <- } Q(s,a) + \alpha[R(s,a) + \gamma \; max_a \, Q(s',a) - Q(s,a)]$$

It is possible for multiple actions to have the same value, for example when a new state is explored and all values are unknown (and all have default value 0). Then, the agent must make a decision based on something other than the value. It could pick an action based on some heuristic, or simply the first action that popped up.

The full row of zeros means that these states have not been discovered, and this can be explained as the track used for training doesn't contain these specific states, we call them rare states, which may appear with non-zero values in another training track. Also it is due to the discretization of speed which may be the value of the speed 0, therefore it is an impossible state to happen, unless it is a starting state.

The zeros which appear in the states containing values means that these actions have not been tried yet and requires more training so it can try all actions with the random policy. The actions with high Q-Values is due to taking the maximum Q-Value over and over until becoming big, this means that this action is the correct one after decreasing epsilon greedy and taking the argmax Q

Q-Learning showed good performance on straight line and normal turns, but it failed miserably in sharp turns as will be shown in the results Section 4.6.

4.6. Results of Q-Learning

This part is containing parameters setting and how we tuned our parameters then the Q-Learning measurements results part.

4.6.1. Parameters Setting

By a lot of experiments, we set this parameter as following in table 4-8:

Table 4-8: Experimental Parameters Values

Parameter	Value
Gamma	0.99
Learning rate	0.01
eta	0.5
Epsilon	0.4

First, we set eta and epsilon with 0.5 and 0.4 respectively to increase the exploration and decreased eta and epsilon each step by 0.000001 to depend much more on the Q-table to achieve exploitation

4.6.2. Q-learning measurement results

Before presenting the figures of the car on the track in TORCS, here is the final measurement results of Q-Learning as shown in the following table 4-9:

Table 4-9: Measurement Results of Q-Learning

Measurement results	Q-learning
Number of updates to build the Model	800,000 updates
Maximum Speed	153 km/h
Time to complete one lap	90 minutes

Next we will represent the results of moving forward in Figure 4.6, then represent making left turn in Figure 4.7 and a sharp left turn in Figure 4.8.

- **Moving in straight line**

As we can see in Figure 4.6 below, our Q-Learning built a Q-table and it was successful at moving forward, but as it approaches a left turn, it goes to the left side preparing for a turn, and this can make it go out of the lane. And this is because it has discretized states and actions, therefore our model isn’t very accurate. But it was able to make an average performance.

It’s also obvious that when it approaches the end of the road, it quickly goes back to the middle of the lane, and it will be clearer in Figure 4.7. Which represents the left turn.

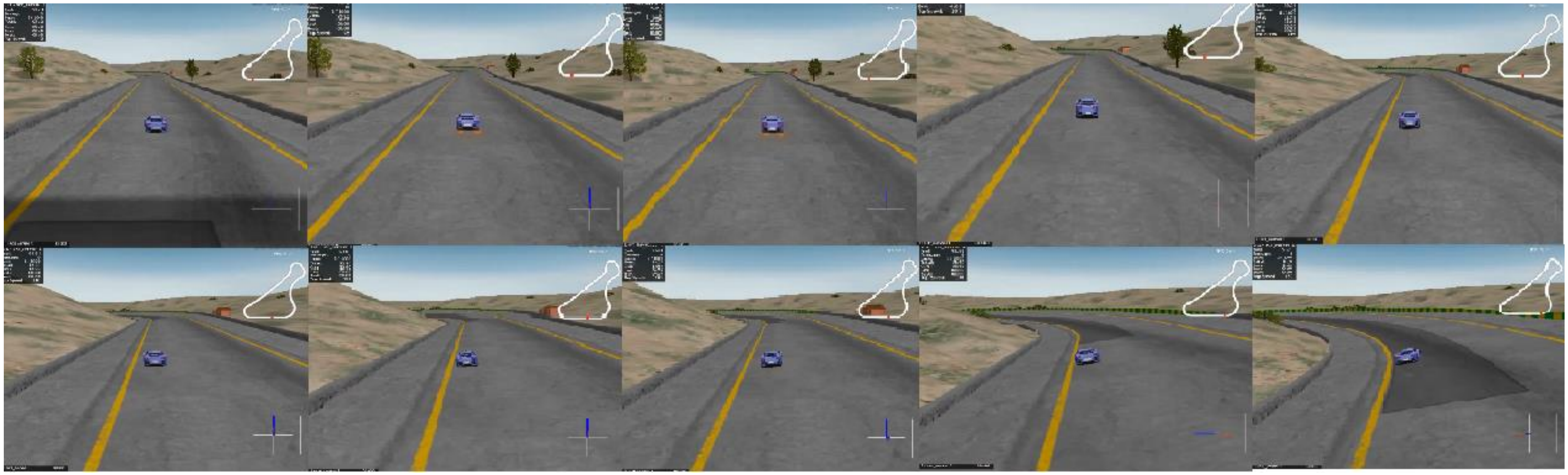


Figure 4-6 Moving in straight line and approaching left turn with Q learning

Performing a left turn then going through a small right turn

From the next Figure 4.7, it is clear that it did a quite good performance on a left turn, but it was hard for it to do a right turn after, that’s why it approached the right edge and then adjusted its track position again.

The middle line is the edge of the black line as shown in the figure above, and these results were taken at a high speed of 120-150

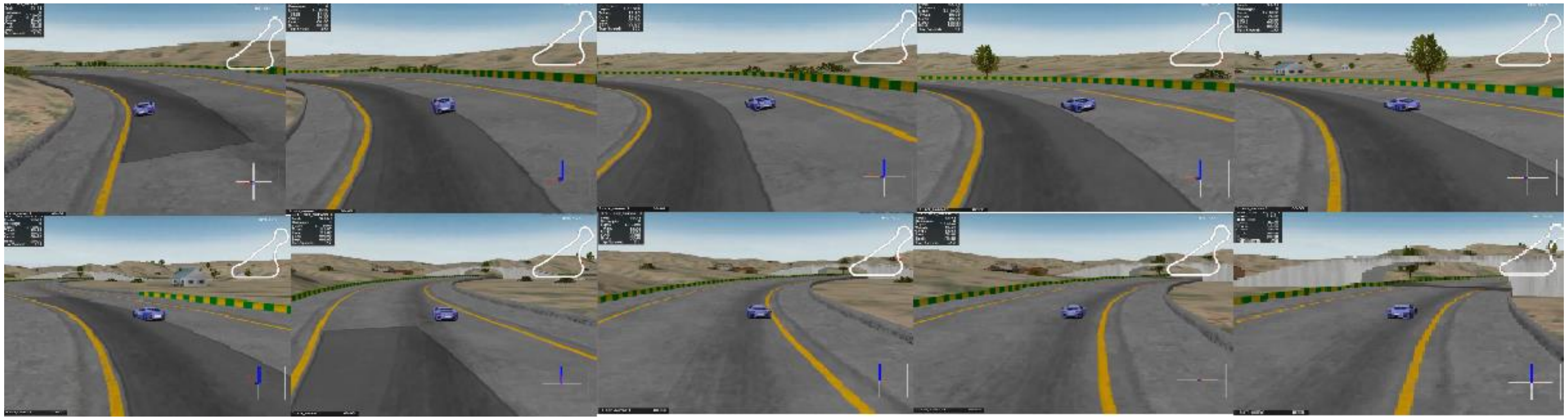


Figure 4-7 Left turn followed by right turn with Q-Learning

Performing a sharp left turn

In the next Figure 4.8 below shows that Q-Learning fails sharp turns. Obviously, it went out of track, but it successfully turned inside again and adjusted its track position.

As we said before Q-Learning is a Discrete Action Algorithm, therefore it’s not very accurate and it might go off the lane as shown in sharp turns.

And this can be explained due to the fact that there are states that the vehicle should take different actions in, but these states are considered as one state due to discretization.

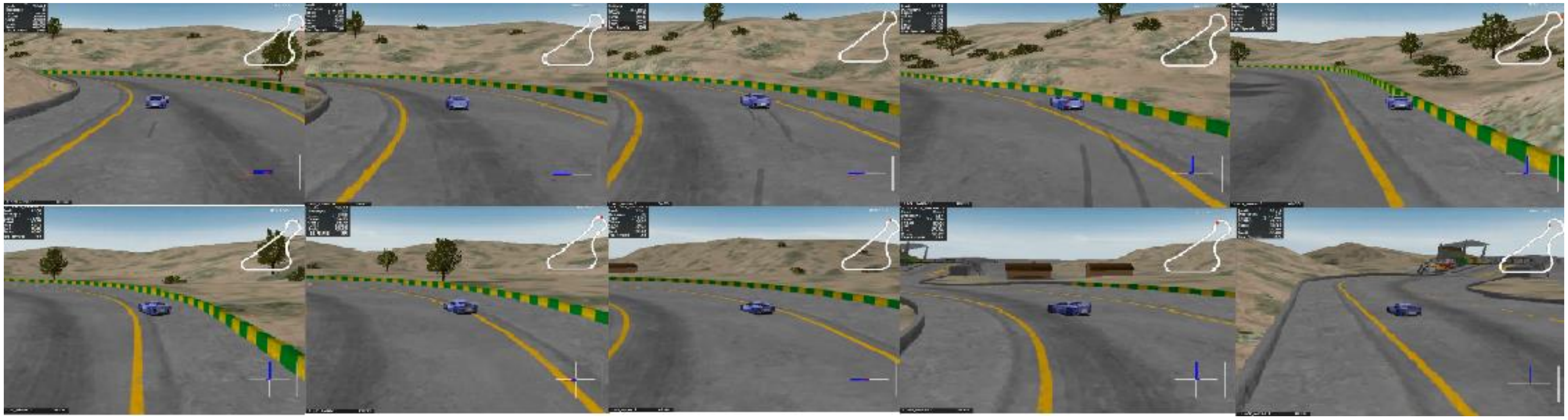


Figure 4-8 Sharp left turn with Q-Learning

4.7. Conclusion

We have explained the simulator used (TORCS) and how do we need to interface with it, also explained the game loop which is essential for running the driver function and how do we interface with the driver which is responsible for moving the vehicle, then gave a sample of the Q-Table and explained how to update it and finally, shown the results obtained.

In the following Chapter, we will implement the Continuous Action Algorithm, specifically, Deep Deterministic Policy Gradient and show its performance on TORCS.

Chapter 5
Implementation of DDPG on TORCS

In this chapter we will explain how we interface with TORCS first, then the methods of implementing Discrete Action algorithm specifically, Q-Learning then explain the Continuous Action algorithm specifically, Deep Deterministic Policy Gradient and show the result of each and compare them.

5.1. DDPG implementation

DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. Policy gradient algorithms utilize a form of policy iteration: they evaluate the policy, and then follow the policy gradient to maximize performance. Since DDPG is off-policy and uses a deterministic target policy, this allows for the use of the Deterministic Policy Gradient theorem. DDPG is an actor-critic algorithm as well; it primarily uses two neural networks, one for the actor and one for the critic. These networks compute action predictions for the current state and generate a temporal-difference (TD) error signal each time step. The input of the actor network is the current state, and the output is a single real value representing an action chosen from a continuous action space. The critic's output is simply the estimated Q-value of the current state and of the action given by the actor. The deterministic policy gradient theorem provides the update rule for the weights of the actor network. The critic network is updated from the gradients obtained from the TD error signal.

In general, training and evaluating the policy and/or value function with thousands of temporally-correlated simulated trajectories leads to the introduction of enormous amounts of variance in your approximation of the true Q-function (the critic). The TD error signal is excellent at compounding the variance introduced by your bad predictions over time.

We used a replay buffer to store the experiences of the agent during training, and then randomly sample experiences to use for learning in order to break up the temporal correlations within different training episodes. This technique is known as experience replay. DDPG uses this. Directly updating the actor and critic neural network weights with the gradients obtained from the TD error signal that was computed from both your replay buffer and the output of the actor and critic networks causes your learning algorithm to diverge (or to not learn at all). It was recently discovered that using a set of target networks to generate the targets for your TD error computation regularizes your learning algorithm and increases stability.

Accordingly, here are the equations for the TD target and the loss function for the critic network. For more details, please refer to the review in Chapter 3.

$$\mathrm{L}_{\mathrm{critic}} \approx \frac{1}{N}\sum_{i=0}^{N}(\mathrm{V}_{\theta}(s_i) - [r_i \cdot \gamma \cdot V(s_i{}')])^2$$

Now, as mentioned above, the weights of the critic network can be updated with the gradients obtained from the loss function. Also, remember that the actor network is updated with the Deterministic Policy Gradient.

$$J \approx \frac{1}{N}\sum_{i=0}^{N}\sum_{s,a\in z_i} R(s,a)$$

We update weights of the critic network

$$W_{k+1} = W_k + \alpha\frac{\partial J}{\partial \theta}$$

Where we discussed in Chapter 3 how to compute this derivative using duct tape approach or by using the log-derivative trick.

5.1.1. Actor Network

We used 2 hidden layers with 150 and 300 hidden units respectively as shown in Figure 5.1. The output consists of 3 continuous actions, **Steering**, which is a single unit with tanh activation function. **Acceleration**, which is a single unit with sigmoid activation function. **Brake**, another single unit with sigmoid activation function, then we updated the Actor Network from the gradients computed from the Critic Network.

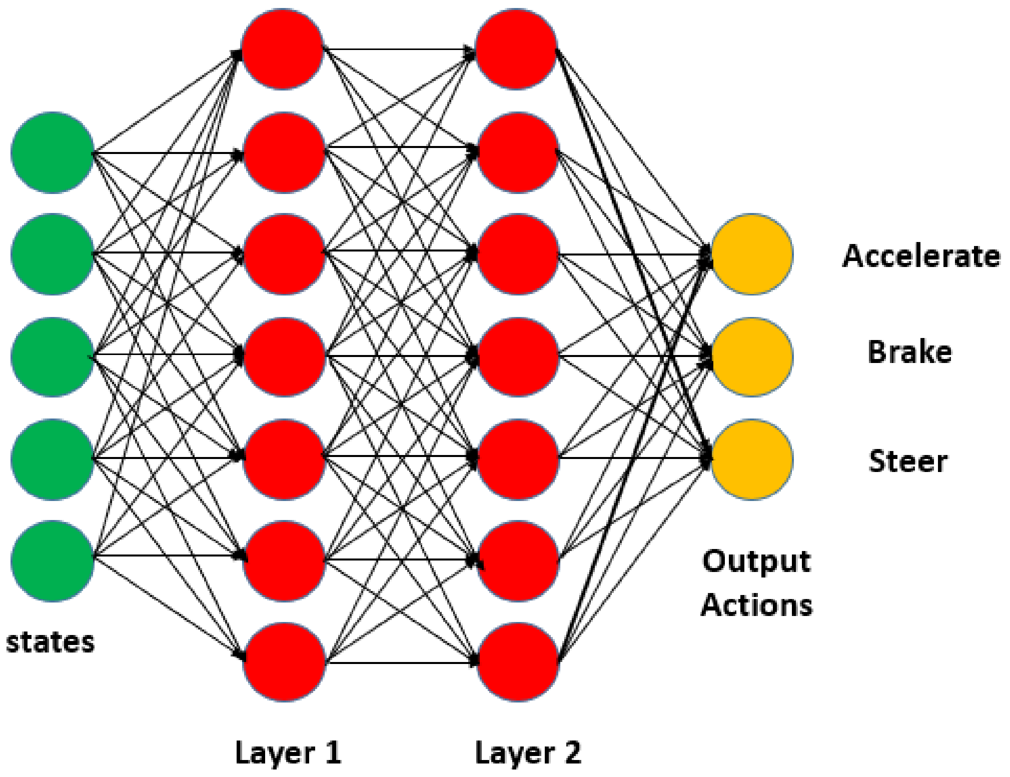


Figure 5-1 Actor Network architecture

5.1.2. Critic Network

We used 2 hidden layers with 150 and 300 hidden units. Also, the critic network takes both the states and action as inputs and outputs the Q-Value. as shown in the following Figure 5.2.

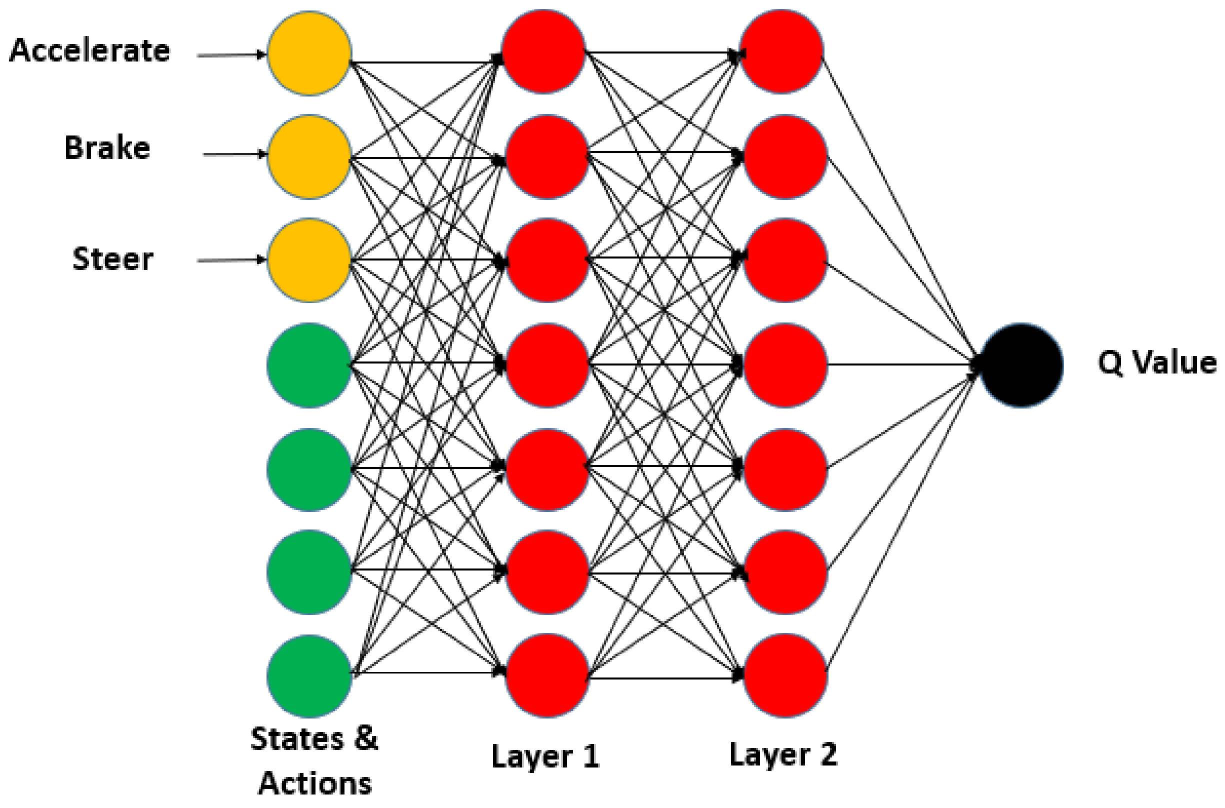


Figure 5-2 critic Network architecture

5.2. DDPG Algorithm

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'}$ and $\theta^{\mu'}$
Initialize Replay buffer R
**For** episode =1 to M do
        Initialize a random process $N$ for action exploration
        Receive initial observation state $s_1$
        **For** t =1 to T do
            Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy
            Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
            Store transition ($s_t$, $a_t$, $r_t$, $s_{t+1}$) in the replay buffer
            Sample a random minibatch of N transitions ($s_t$, $a_t$, $r_t$, $s_{t+1}$) from buffer
            Set target $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})\ |\theta^{\mu'})$
            Update Critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(S_i, a_i|\ \theta^Q))^2$
            Update the Actor policy using the sampled policy gradient:
                        $\nabla_{\theta^\mu} J = \frac{1}{N}\sum_i \nabla_a\ Q(S_i, a_i|\ \theta^Q) * \nabla_{\theta^\mu}\ \mu(s_t|\theta^\mu)$
            Update the target networks:
                        $\theta^{Q'} \leftarrow \tau\,\theta^Q + (1-\tau)\theta^{Q'}$
                        $\theta^{\mu'} \leftarrow \tau\,\theta^{\mu} + (1-\tau)\theta^{\mu'}$
        **End for**
**End for**
```

Directly implementing Q learning with neural networks proved to be unstable in many environments. Since the Eval`s critic network being updated is also used in calculating the target value, the Q update is prone to divergence.

We modified the actor-critic and used “soft” target updates, rather than directly copying the weights.

$$\theta^{Q'} \leftarrow \tau\,\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\,\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

We create a copy of the actor and critic networks, target`s critic and target`s actor that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks.

This means that the target values are constrained to change slowly, greatly improving the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist. We found that having both a target policy and target Q-value was required to have stable targets.

In order to consistently train the critic without divergence. This may slow learning, since the target network delays the propagation of value estimations. However, in practice we found this was greatly outweighed by the stability of learning.

A major challenge of learning in continuous action spaces is exploration. An advantage of off-policies algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. We constructed an exploration policy μ' by adding noise sampled from a noise process N to our actor policy

$$a_t = \mu(s_t|\theta^{\mu}) + N_t$$

5.3. Flow of the implementation of DDPG

We have used a framework in python called Tensorflow, which can represent the flow graph of the neural networks used in a convenient way. Figure 5.3 represents the tensorflow graph of the DDPG model is used used and will be explained in details.

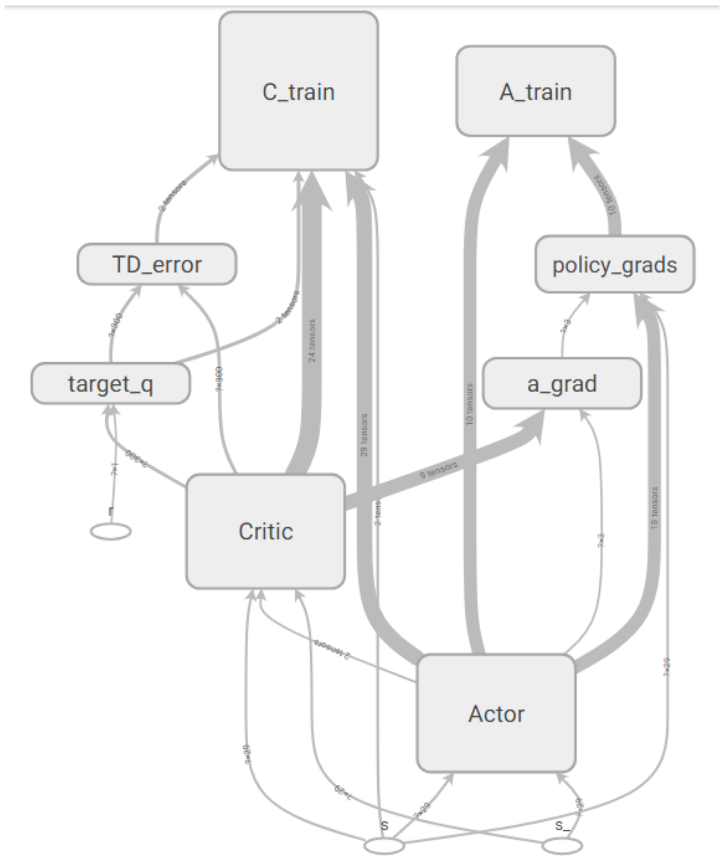


Figure 5-3 Tensorflow graph of DDPG

The graph blocks:

Actor block

Consists of 2 networks: evaluation and target, they both take current state (s) and next state (s_) and feed them into the actor network and then outputs actions. These actions are given to the Critic block

Critic block

Take the actions from Actor block and the states and it also consists of evaluation and target networks- next to the current state (s) to output the Q-value, then with back propagation we compute the policy gradients (a_grad block), which will be used in policy_grads block to train the actor target network in A_train block.

a_grad block

Block which contain the derivatives of the evaluation Q value with respect to action.

policy_grads block

This block is used to update the weights of the actor target network. And are computed from the TD_error block

A_train block

Block which trains the Actor neural network

TD_error block

Takes target_q which is simply the Q-Value of the actual next state and the evaluation q value in evaluation network. And this is done by reducing the loss function in C_train block.

C_train block

Block which minimizes the loss between the Target and Q value of the current state

The reason of putting 2 networks is to make actions and evaluate with one network and update weights of the target network to become close to the optimal policy. For more details please refer to Chapter 3.

5.4. Results of DDPG

In the below table 5-1 we will show the results of the DDPG measurements as follow:

Table 5-1: DDPG Measurement Results

Measurement results	DDPG Algorithm
Number of updates to build the Model	300,000 updates
Maximum Speed	178 km/h
Time to complete one lap	120 minutes

Then we will present the straight-line performance in Figure 5.4, then performing a left turn in Figure 5.5 and finally a sharp left turn in Figure 5.6.

Moving in straight line

The result shown in the next Figure 5.4, shows that DDPG outperforms Q-Learning in straight line lane, and it moves almost perfectly in straight line. This is due to the fact that it takes continuous values of all sensors and outputs also continuous values of actions.

The model isn’t very accurate, because the training time wasn’t enough, and the low number of layers. This is due to the lack of GPU, as we used CPU only in training.

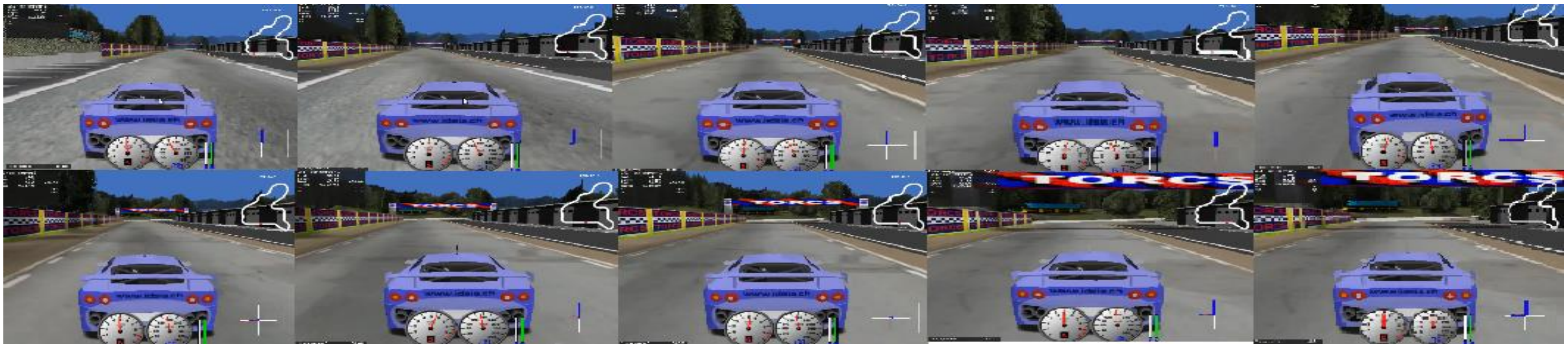


Figure 5-4 Moving in straight line with DDPG

Performing a left turn

As we can see in the next Figure 5.5, DDPG does a very good performance on left turns, it has slight errors as we said before due to the lack of neural network layers and training time. But overall it is better than Q-Learning, as Q-Learning keeps moving left and right alternatively due to discretization.

We will look into the sharp turns in the next figure followed by a sharper turn, then compare it with Q-Learning performance.

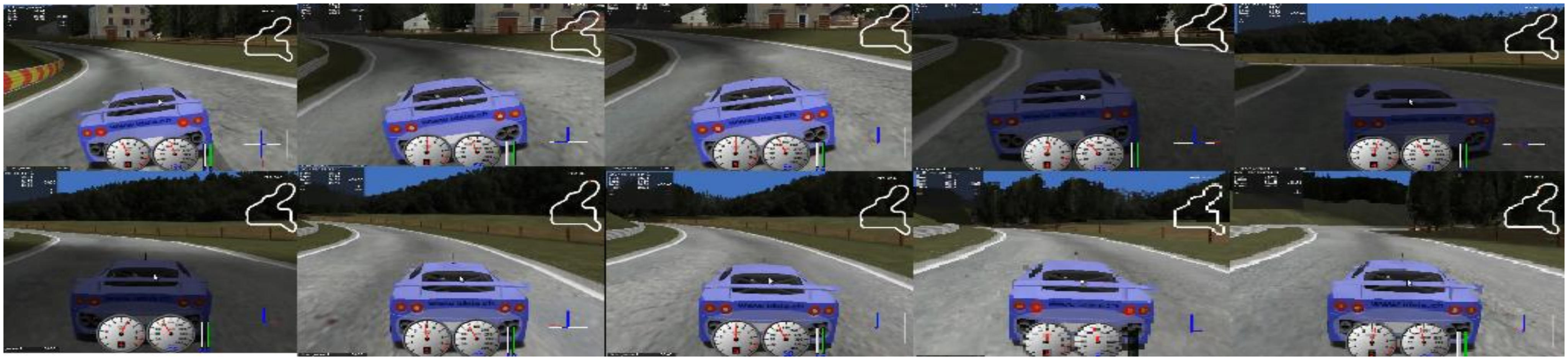


Figure 5-5 Performing a normal left turn with DDPG

Performing a sharp left turn

The first and second row in Figure 5.6 resembles the sharp left turn, and it showed an impressive performance as it didn’t go out of the lane unlike Q-Learning that failed and went off the track.

The third row in Figure 5.6 resembles a sharper left turn than first and second row, and the performance was truly exceptional, as the vehicle learnt to slow down a bit on very sharp turns and to take correct actions (of course not very accurate, but still acceptable performance). This performance wouldn’t be achieved in Q-Learning.

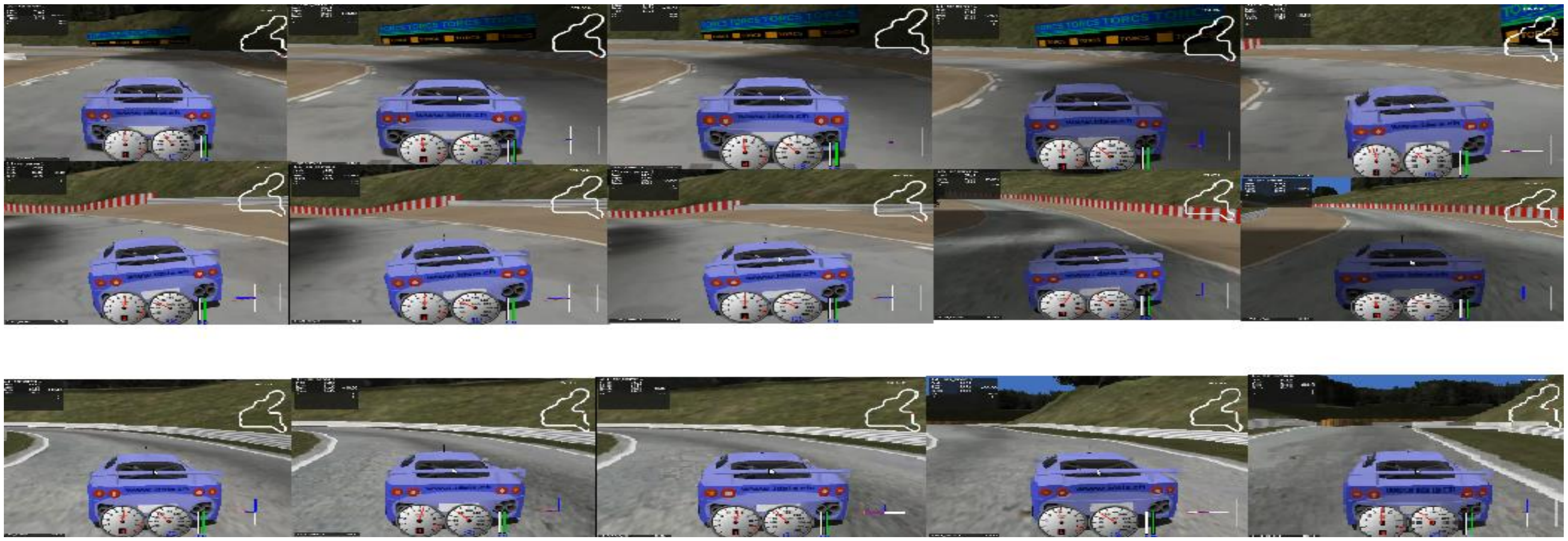


Figure 5-6 Performing Sharp left turn with DDPG

5.5. Conclusion and Comparison between DDPG and Q-Learning

In this part we will try to discuss the differences between DDPG and Q-Learning during the training and using some factors as shown in table 5-2.

Table 5-2: Comparison between DDPG and Q-Learning

Measurement results	Q-learning	DDPG Algorithm
Number of updates to build the Model	800,000 updates	300,000 updates
Maximum Speed	153 km/h	178 km/h
Time to complete one lap	90 minutes	120 minutes

As we saw in the above table DDPG took less number of updates to build the final model. DDPG less by 62.5% than Q-Learning algorithm in number of updates also the speed of the car in DDPG is faster than the car in Q-Leaning by 14%.

The final and the most important note that the DDPG took more time to complete one lab during the training and this is due to the informed action part of Q-learning which help the car and work as a guide so it can complete one lab in less time than the DDPG.

Chapter 6
Conclusions and Future Works

In this chapter, we will introduce with two parts which are the conclusion in Section 6.1 in this work in addition to the future research works which could help in the extension of this project.

6.1. Conclusions

Autonomous driving with Lane Keeping Assist feature is a very difficult problem to be solved, especially with algorithms that are still in a research area such as Reinforcement Learning algorithms. It might seem easy to apply them with a simulator, but in real life it is totally different. This is due to the numerous factors that aren't considered and which also can affect the vehicle, such as: Climate changes, pedestrians, congestion, traffic lights.

We applied Reinforcement Learning on TORCS Simulator in 2 different approaches: Discrete Action Algorithm, specifically Q-Learning and Continuous Action Algorithm, specifically Deep Deterministic Policy Gradient (DDPG).

1- In case of Q-Learning, the discretized Q-Table was able to drive in a track in TORCS successfully without being crashed. But it also couldn't do sharp turns efficiently and it drives off the lane and returns back.
2- It is impossible to make a full look-up table with Q-Values for all sensors and all actions that's why we reduced the number of states by reducing the number of sensors.
3- In case of DDPG there was no need to reduce any of the states, and we took all continuous values of sensors using neural networks. That's why it was more accurate as neural network has better stability.

Then we compared the performance between them, and it was clear that

4- The DDPG was smoother and more stable than Q-Learning. This means that the Continuous Action Algorithm is more suitable for our problem and can be used for future works and enhancements.
5- DDPG has faster convergence than Q-Learning as it finished training in less number of episodes than Q-Learning.
6- DDPG has found to be faster than Q-Learning which means that it can arrive to a destination fast.

A few limitations to our approach remain. Most notably, as with most model-free reinforcement approaches, DDPG requires a large number of training episodes to find solutions. However, we believe that a robust model-free approach may be an important component of larger systems which may attack these limitations

6.2. Future Works

We can extend our work by doing any of the following:

1. We can use Data Aggregation way in order to have a more generalized model for Autonomous Driving.
2. General measures in reinforcement learning for finding a better policy include a decreasing learning rate and a decreasing exploration rate.
3. DDPG can be used on a real car.
4. Instead of using TORCS sensors we can use sensors coming from screenshots of the road taken by a camera and do preprocessing.
5. We can use more sensors and make it more accurate.
6. We can enhance LKA and make it avoid obstacles with retraining and adding the sensors needed.
7. We will apply Autonomous Driving plus Lane Keeping Assist feature on Public transportation in order to be fully automated and prevent any accidents may be happened.

References

[1] **Abbeel, P., & Ng, A. Y.,** "Apprenticeship learning via inverse reinforcement learning", (2004).

[2] **Alexander Panin, Pavel Shvechikov, "**Practical Reinforcement Learning" course at Coursera.org, Created by National Research University Higher School of Economics, Yandex, (2018).

[3] **Daniel Karavolos**, "Q-learning with heuristic exploration in Simulated Car Racing", A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Artificial Intelligence at the University of Amsterdam, The Netherlands, (August 2013).

[4] **David Silver,** Introduction to Reinforcement Learning at UCL University, (2015).

[5] **El Sallab, A., Abdou, M., Perot, E., and Yogamani, S.,** "Deep Reinforcement Learning framework for Autonomous Driving", (2017).

[6] **Lex Weaver, Nigel Tao, "**The Optimal Reward Baseline for Gradient·Based Reinforcement Learning", Department of Computer Science Australian National University, (Jan 2013).

[7] **Mohammed Abdou Tolba, "**Autonomous Driving Deep Reinforcement Learning", A Thesis Submitted to the Faculty of Engineering at Cairo University in Partial Fulfillment of the Requirements for the Degree of Master of Science in Electronics and Communications Engineering, (2017).

[8] **Morvan,** "Deep Reinforcement Learning Tutorials" at YouTube channel https://www.youtube.com/user/MorvanZhou

[9] **Ng, A. Y., "**Introduction to Machine Learning" course at Coursera.org, (2016).

[10] **Siraj Raval, "**How to use Q-Learning, Deep Q-Learning in Video Games Easily" at YouTube https://www.youtube.com/watch?v=A5eihauRQvo

[11] **Sutton, R. S. and Barto, A. G.** "Reinforcement Learning: An Introduction". MIT Press, Cambridge, MA (1998).

[12] **Tingting Zhao, Hirotaka Hachiya, Gang Niu, and Masashi Sugiyama, "**Analysis and Improvement of Policy Gradient Estimation", Tokyo Institute of Technology, (2017).